

26. Using VPI routines

Clause 26 and Clause 27 specify the Verilog Procedural Interface (VPI) for the Verilog HDL. This clause describes how the VPI routines are used, and Section 27 defines each of the routines in alphabetical order.

26.1 VPI system tasks and functions

User defined system tasks and functions are created using the routine **vpi_register_systf()** (see 27.34). The registration of system tasks must occur prior to elaboration or the resolution of references.

The intended use model would be to place a reference to a routine within the **vlog_startup_routines[]** array. This routine would register all user defined system tasks and functions when it is called.

VPI system tasks have *compiletf*, *size tf*, and *calltf* routines which have the same use model as the corresponding *checktf*, *size tf* and *calltf* routines in the TF interface mechanism for user defined system tasks and functions (refer to Clause 21). The functionality provided in the TF interface mechanism for the *misctf* routine is supported via a set of callbacks, which can be registered using **vpi_register_cb()**.

26.2 The VPI interface

The VPI interface provides routines that allow Verilog product users to access information contained in a Verilog design, and that allow facilities to interact dynamically with a software product. Applications of the VPI interface can include delay calculators and annotators, connecting a Verilog simulator with other simulation and CAE systems, and customized debugging tasks.

The functions of the VPI interface can be grouped into two main areas:

- Dynamic software product interaction using VPI callbacks
- Access to Verilog HDL objects and simulation specific objects

26.2.1 VPI callbacks

Dynamic software product interaction shall be accomplished with a registered callback mechanism. VPI callbacks shall allow a user to request that a Verilog HDL software product, such as a logic simulator, call a user-defined application when a specific activity occurs. For example, the user can request that the user application *my_monitor()* be called when a particular net changes value, or that *my_cleanup()* be called when the software product execution has completed.

The VPI callback facility shall provide the user with the means to interact dynamically with a software product, detecting the occurrence of value changes, advancement of time, end of simulation, etc. This feature allows applications such as integration with other simulation systems, specialized timing checks, complex debugging features, etc.

The reasons for which callbacks shall be provided can be separated into four categories:

- Simulation event* (e.g., a value change on a net or a behavioral statement execution)
- Simulation time* (e.g., the end of a time queue or after certain amount of time)
- Simulator action/feature* (e.g., the end of compile, end of simulation, restart, or enter interactive mode)
- User-defined system task or function execution*

VPI callbacks shall be registered by the user with the functions **vpi_register_cb()** and **vpi_register_systf()**. These routines indicate the specific reason for the callback, the application to be called, and what system and user data shall be passed to the callback application when the callback occurs. A facility is also provided to call the callback functions when a Verilog HDL product is first invoked. A primary use of this facility shall be for registration of user-defined system tasks and functions.

26.2.2 VPI access to Verilog HDL objects and simulation objects

Accessible Verilog HDL objects and simulation objects and their relationships and properties are described using data model diagrams. These diagrams are presented in 26.6. The data model diagrams indicate the routines and constants that are required to access and manipulate objects within an application environment. An associated set of routines to access these objects is defined in Clause 27.

The VPI interface also includes a set of utility routines for functions such as handle comparison, file handling, and redirected printing, which are described in Table 211.

VPI routines provide access to objects in an *instantiated* Verilog design. An instantiated design is one where each instance of an object is uniquely accessible. For instance, if a module *m* contains wire *w* and is instantiated twice as *m1* and *m2*, then *m1.w* and *m2.w* are two distinct objects, each with its own set of related objects and properties.

The VPI interface is designed as a *simulation* interface, with access to both Verilog HDL objects and specific simulation objects. This simulation interface is different from a hierarchical language interface, which would provide access to HDL information but would not provide information about simulation objects.

has not been
implemented
in ModelSim 5.8 yet.

26.2.3 Error handling

To determine if an error occurred, the routine **vpi_chk_error()** shall be provided. The **vpi_chk_error()** routine shall return a nonzero value if an error occurred in the previously called VPI routine. Callbacks can be set up for when an error occurs as well. The **vpi_chk_error()** routine can provide detailed information about the error.

26.2.4 Function availability

Certain features of the VPI interface must occur early in the execution of a tool. In order to allow this process to occur in an orderly manner, some functionality must be restricted in these early stages. Specifically, when the routines within the **vlog_startup_routines[]** array are executed, there is very little functionality available. Only two routines can be called at this time:

vpi_register_systf()
vpi_register_cb()

In addition, the **vpi_register_cb()** routine can only be called for the following reasons:

cbEndOfCompile
cbStartOfSimulation
cbEndOfSimulation
cbUnresolvedSystf
cbError
cbPLIError

Refer to 27.34 for a further explanation of the use of the **vlog_startup_routines[]** array.

The next earliest phase is when the **sizetf** routines are called for the user defined system functions. At this phase, no additional access is permitted. After the **sizetf** routines are called, the routines registered for reason **cbEndOfCompile** are called. At this point, and continuing until the tool has finished execution, all functionality is available.

26.2.5 Traversing expressions

The VPI routines provide access to any expression which can be written in the HDL. Dealing with these expressions can be complex, since very complex expressions can be written in the HDL. Expressions with

multiple operands will result in a handle of type **vpiOperation**. To determine how many operands, access the property **vpiOpType**. This operation will be evaluated after its subexpressions. Therefore, it has the least precedence in the expression.

An example of a routine which traverses an entire complex expression is listed below:

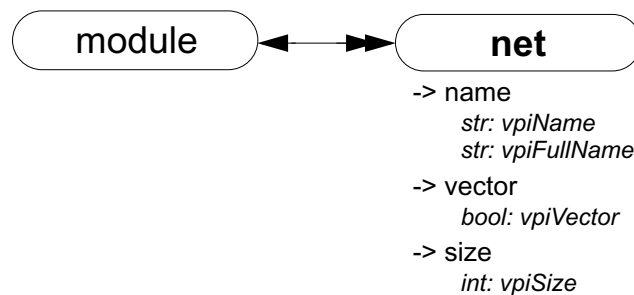
```
void traverseExpr(vpiHandle expr)
{
    vpiHandle subExprI, subExprH;

    switch (vpi_get(vpiExpr, expr))
    {
        case vpiOperation:
            subExprI = vpi_iterate(vpiOperand, expr);
            if (subExprI)
                while (subExprH = vpi_scan(subExprI))
                    traverseExpr(subExprH);
            /* else it's of op type vpiNullOp */
            break;
        default:
            /* Do whatever to the leaf object. */
            break;
    }
}
```

26.3 VPI object classifications

VPI objects are classified using data model diagrams. These diagrams provide a graphical representation of those objects within a Verilog design to which the VPI routines shall provide access. The diagrams shall show the relationships between objects and the properties of each object. Objects with sufficient commonality are placed in groups. Group relationships and properties apply to all the objects in the group.

As an example, this simplified diagram shows that there is a *one-to-many relationship* from objects of type **module** to objects of type **net**, and a *one-to-one relationship* from objects of type **net** to objects of type **module**. Objects of type **net** have properties **vpiName**, **vpiVector**, and **vpiSize**, with data types string, boolean, and integer respectively.



The VPI data model diagrams are presented in 26.6.

For object relationships (unless a special tag is shown in the diagram), the type used for access is determined by adding **vpi** to the beginning of the word within the enclosure with each word's first letter being a capital. Using the above example, if an application has a handle to a net, and wants to go to the module instance where the net is defined, the call would be:

```
modH = vpi_handle(vpiModule, netH);
```

where `netH` is a handle to the net. As another example, to access a named event object, use the type `vpiNamedEvent`.

26.3.1 Accessing object relationships and properties

The VPI interface defines the C data type of `vpiHandle`. All objects are manipulated via a `vpiHandle` variable. Object handles can be accessed from a relationship with another object, or from a hierarchical name, as the following example demonstrates:

```
vpiHandle net;
net = vpi_handle_by_name("top.m1.w1", NULL);
```

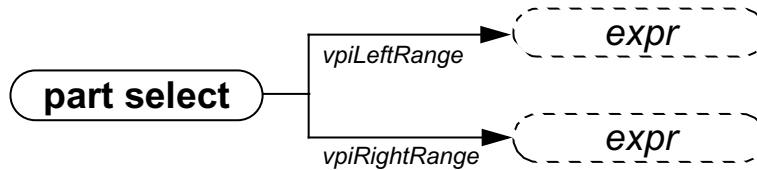
This example call retrieves a handle to wire `top.m1.w1` and assigns it to the `vpiHandle` variable `net`. The `NULL` second argument directs the routine to search for the name from the top level of the design.

The VPI interface provides generic functions for tasks, such as traversing relationships and determining property values. One-to-one relationships are traversed with routine `vpi_handle()`. In the following example, the module that contains `net` is derived from a handle to that net:

```
vpiHandle net, mod;
net = vpi_handle_by_name("top.m1.w1", NULL);
mod = vpi_handle(vpiModule, net);
```

The call to `vpi_handle()` in the above example shall return a handle to module `top.m1`.

Sometimes it is necessary to access a class of objects which do not have a name, or whose name is ambiguous with another class of objects which can be accessed from the reference handle. *Tags* are used in this situation.



In this example, the tags `vpiLeftRange` and `vpiRightRange` are used to access the expressions which make up the range of the part select. These tags are used *instead* of `vpiExpr` to get to the expressions. Without the tags, the VPI interface would not know which expression should be accessed. For example:

```
vpi_handle(vpiExpr, part_select_handle)
```

would be illegal when the reference handle (`part_select_handle`) is a handle to a part select, because the part select can refer to two expressions, a left-range and a right-range.

Properties of objects shall be derived with routines in the `vpi_get` family. The routine `vpi_get()` returns integer and boolean properties. Integer and boolean properties shall be defined to be of type `PLI_INT32`. For boolean properties, a value of `1` shall represent **TRUE** and a value of `0` shall represent **FALSE**. The routine `vpi_get_str()` accesses string properties. String properties shall be defined to be of type `PLI_BYTE8 *`. For example: to retrieve a pointer to the full hierarchical name of the object referenced by handle `mod`, the following call would be made:

```
PLI_BYTE8 *name = vpi_get_str(vpiFullName, mod);
```

In the above example, the pointer `name` shall now point to the string `"top.m1"`.

One-to-many relationships are traversed with an iteration mechanism. The routine **vpi_iterate()** creates an object of type **vpiIterator**, which is then passed to the routine **vpi_scan()** to traverse the desired objects. In the following example, each net in module `top.m1` is displayed:

```
vpiHandle itr;
itr = vpi_iterate(vpiNet,mod);
while (net = vpi_scan(itr) )
    vpi_printf("\t%s\n", vpi_get_str(vpiFullName, net) );
```

As the above examples illustrate, the routine naming convention is a *vpi* prefix with `_` word delimiters (with the exception of callback-related defined values, which use the *cb* prefix). Macro-defined types and properties have the *vpi* prefix, and they use capitalization for word delimiters.

The routines for traversing Verilog HDL structures and accessing objects are described in Clause 27.

26.3.2 Object type properties

All objects have a **vpiType** property, which is not shown in the data model diagrams.

-> type
int: vpiType

Using **vpi_get(vpiType, <object_handle>)** returns an integer constant which represents the type of the object.

Using **vpi_get_str(vpiType, <object_handle>)** returns a pointer to a string containing the name of the type constant. The name of the type constant is derived from the name of the object as it is shown in the data model diagram (refer to 26.3 for a description of how type constant names are derived from object names).

Some objects have additional type properties which are shown in the data model diagrams; **vpiDelayType**, **vpiNetType**, **vpiOpType**, **vpiPrimType**, **vpiResolvedNetType** and **vpiTchkType**. Using **vpi_get(<type_property>, <object_handle>)** returns an integer constant which represents the additional type of the object. Refer to `vpi_user.h` in Annex G for the types which can be returned for these additional type properties. The constant names of the types returned for these additional type properties can be accessed using **vpi_get_str()**.

26.3.3 Object file and line properties

Most objects have two location properties, which are not shown in the data model diagrams:

-> location
int: vpiLineNo
str: vpiFile

The properties **vpiLineNo** and **vpiFile** can be affected by the **line** and **file** compiler directives. See Section 19 for more details on these compiler directives. These properties are applicable to every object that corresponds to some object within the HDL. The exceptions are objects of type:

```
vpiCallback
vpiDelayTerm
vpiDelayDevice
vpiInterModPath
vpiIterator
vpiTimeQueue
```

26.3.4 Delays and values

Most properties are of type integer, boolean, or string. Delay and logic value properties, however, are more complex and require specialized routines and associated structures. The routines **vpi_get_delays()** and **vpi_put_delays()** use structure pointers, where the structure contains the pertinent information about delays. Similarly, simulation values are also handled with the routines **vpi_get_value()** and **vpi_put_value()**, along with an associated set of structures.

The routines, C structures, and some examples for handling delays and logic values are presented in Clause 27. See 27.14 for **vpi_get_value()**, 27.32 for **vpi_put_value()**, 27.9 for **vpi_get_delays()**, and 27.30 for **vpi_put_delays()**.

Nets, primitives, module paths, timing checks, and continuous assignments can have delays specified within the HDL. Additional delays may exist, such as module input port delays or inter-module path delays, that do not appear within the HDL. To access the delay expressions that are specified within the HDL, use the method **vpiDelay**. These expressions shall be either an expression that evaluates to a constant if there is only one delay specified, or an operation if there are more than one delay specified. If multiple delays are specified, then the operation's **vpiOpType** shall be **vpiListOp**. To access the actual delays being used by the tool, use the routine **vpi_get_delays()** on any of these objects.

26.4 List of VPI routines by functional category

The VPI routines can be divided into groups based on primary functionality.

- VPI routines for simulation-related callbacks
- VPI routines for system task/function callbacks
- VPI routines for traversing Verilog HDL hierarchy
- VPI routines for accessing properties of objects
- VPI routines for accessing objects from properties
- VPI routines for delay processing
- VPI routines for logic and strength value processing
- VPI routines for simulation time processing
- VPI routines for miscellaneous utilities

Table 203 through Table 211 list the VPI routines by major category. Clause 27 defines each of the VPI routines, listed in alphabetical order.

Table 203—VPI routines for simulation related callbacks

To	Use
Register a simulation-related callback	vpi_register_cb()
Remove a simulation-related callback	vpi_remove_cb()
Get information about a simulation-related callback	vpi_get_cb_info()

Table 204—VPI routines for system task/function callbacks

To	Use
Register a system task/function callback	vpi_register_systf()
Get information about a system task/function callback	vpi_get_systf_info()

Table 205—VPI routines for traversing Verilog HDL hierarchy

To	Use
Obtain a handle for an object with a one-to-one relationship	vpi_handle()
Obtain handles for objects in a one-to-many relationship	vpi_iterate() vpi_scan()
Obtain a handle for an object in a many-to-one relationship	vpi_handle_multi()

Table 206—VPI routines for accessing properties of objects

To	Use
Get the value of objects with types of <code>int</code> or <code>bool</code>	vpi_get()
Get the value of objects with types of <code>string</code>	vpi_get_str()

Table 207—VPI routines for accessing objects from properties

To	Use
Obtain a handle for a named object	vpi_handle_by_name()
Obtain a handle for an indexed object	vpi_handle_by_index()
Obtain a handle to a word or bit in an array	vpi_handle_by_multi_index()

Table 208—VPI routines for delay processing

To	Use
Retrieve delays or timing limits of an object	vpi_get_delays()
Write delays or timing limits to an object	vpi_put_delays()

Table 209—VPI routines for logic and strength value processing

To	Use
Retrieve logic value or strength value of an object	vpi_get_value()
Write logic value or strength value to an object	vpi_put_value()

Table 210—VPI routines for simulation time processing

To	Use
Find the current simulation time or the scheduled time of future events	vpi_get_time()

Table 211—VPI routines for miscellaneous utilities

To	Use
Write to the output channel of the software product which invoked the PLI application and the current log file	vpi_printf()
Write to the output channel of the software product which invoked the PLI application and the current log file using varargs	vpi_vprintf()
Flush data from the current simulator output buffers	vpi_flush()
Open a file for writing	vpi_mcd_open()
Close one or more files	vpi_mcd_close()
Write to one or more files	vpi_mcd_printf()
Write to one or more open files using varargs	vpi_mcd_vprintf()
Flush data from a given MCD output buffer	vpi_mcd_flush()
Retrieve the name of an open file	vpi_mcd_name()
Retrieve data about product invocation options	vpi_get_vlog_info()
See if two handles refer to the same object	vpi_compare_objects()
Obtain error status and error information about the previous call to a VPI routine	vpi_chk_error()
Free memory allocated by VPI routines	vpi_free_object()
Add user-allocated storage to application saved data	vpi_put_data()
Retrieve user-allocated storage from application saved data	vpi_get_data()
Store user data in VPI work area	vpi_put_userdata()
Retrieve user data from VPI work area	vpi_get_userdata()
Control simulation execution (stop, finish, etc.)	vpi_sim_control()

26.5 Key to data model diagrams

This subsection contains the keys to the symbols used in the data model diagrams. Keys are provided for objects and classes, traversing relationships, and accessing properties.

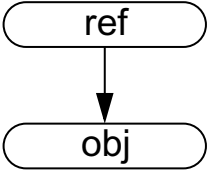
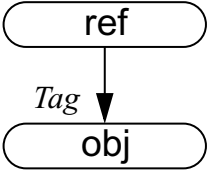
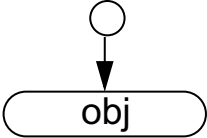
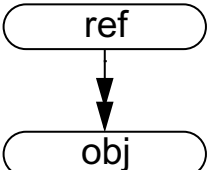
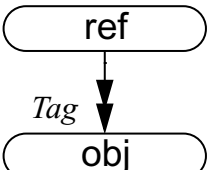
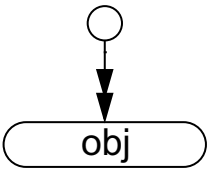
26.5.1 Diagram key for objects and classes

	<p>Object Definition:</p> <p>Bold letters in a solid enclosure indicate an object definition. The properties of the object are defined in this location.</p>
	<p>Object Reference:</p> <p>Normal letters in a solid enclosure indicate an object reference.</p>
	<p>Class Definition:</p> <p><i>Bold italic</i> letters in a dotted enclosure indicate a class definition, where the class groups other objects and classes. Properties of the class are defined in this location. The class definition can contain an object definition.</p>
	<p>Class Reference:</p> <p><i>Italic</i> letters in a dotted enclosure indicate a class reference.</p>
	<p>Unnamed Class:</p> <p>A dotted enclosure with no name is an unnamed class. It is sometimes convenient to group objects although they shall not be referenced as a group elsewhere, so a name is not indicated.</p>

26.5.2 Diagram key for accessing properties

 -> vector <i>bool: vpiVector</i> -> size <i>int: vpiSize</i>	<p>Integer and boolean properties are accessed with the routine vpi_get(). These properties are of type PLI_INT32.</p> <p>Example: Given handle <code>obj_h</code> to an object of type vpiObj, test if the object is a vector, and get the size of the object.</p> <pre>PLI_INT32 vect_flag = vpi_get(vpivector, obj_h); PLI_INT32 size = vpi_get(vpiSize, obj_h);</pre>
 -> name <i>str: vpiName</i> <i>str: vpiFullName</i>	<p>String properties are accessed with routine vpi_get_str(). String properties are of type PLI_BYTE8 *.</p> <p>Example:</p> <pre>PLI_BYTE8 *name = vpi_get_str(vpiName, obj_h);</pre>
 -> complex <i>func1()</i> <i>func2()</i>	<p>Complex properties for time and logic value are accessed with the indicated routines. See the descriptions of the routines for usage.</p>

26.5.3 Diagram key for traversing relationships

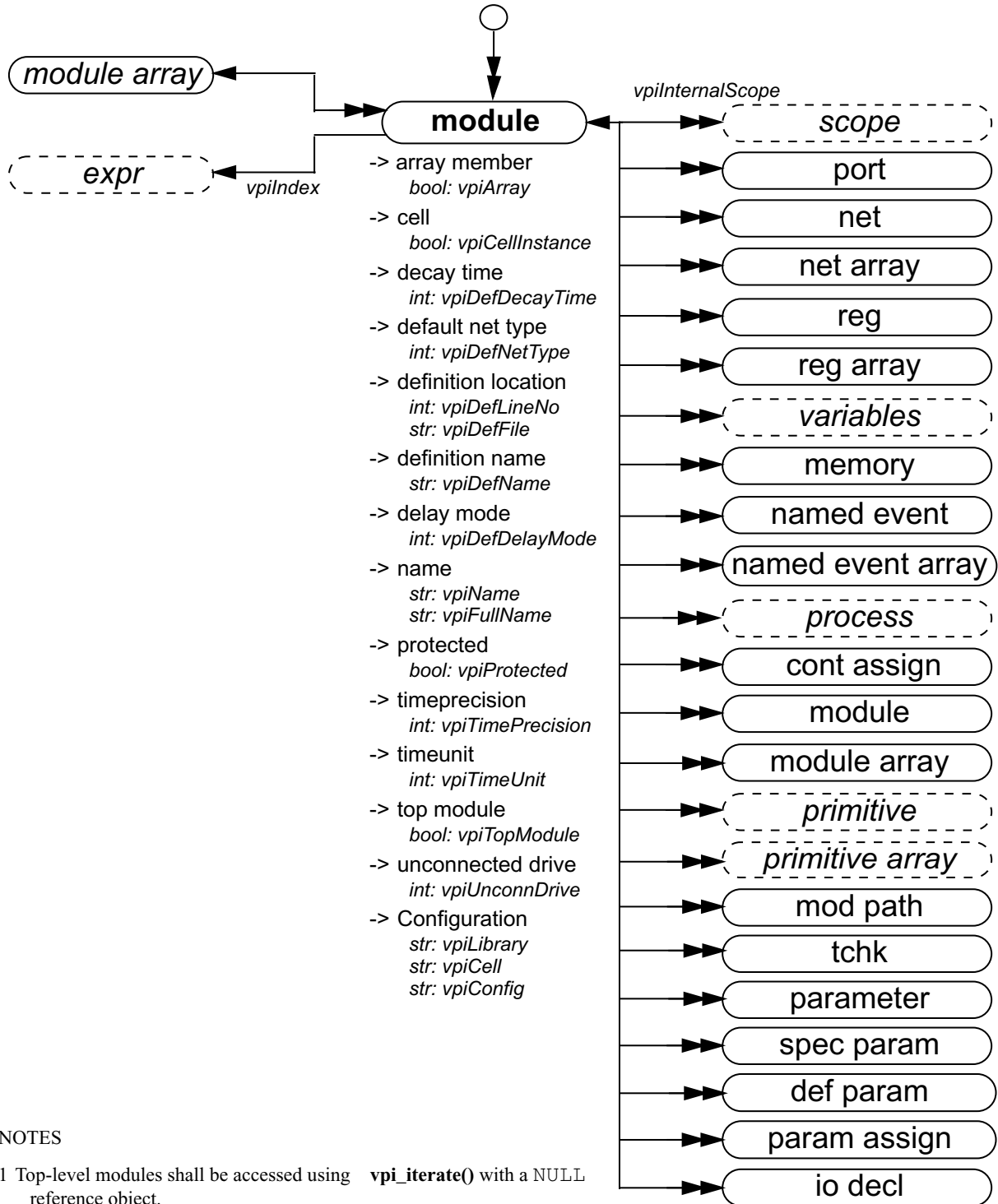
	<p>A single arrow indicates a <i>one-to-one</i> relationship accessed with the routine vpi_handle().</p> <p>Example: Given vpiHandle variable <code>ref_h</code> of type <code>ref</code>, access <code>obj_h</code> of type <code>Obj</code>:</p> <pre>obj_h = vpi_handle(Obj, ref_h);</pre>
	<p>A tagged <i>one-to-one</i> relationship is traversed similarly, using <i>Tag</i> instead of <i>Obj</i>:</p> <p>Example:</p> <pre>obj_h = vpi_handle(Tag, ref_h);</pre>
	<p>A <i>one-to-one</i> relationship which originates from a circle is traversed using <code>NULL</code> for the <code>ref_h</code>:</p> <p>Example:</p>
	<p>A double arrow indicates a <i>one-to-many</i> relationship accessed with the routine vpi_scan().</p> <p>Example: Given vpiHandle variable <code>ref_h</code> of type <code>ref</code>, scan objects of type <code>Obj</code>:</p> <pre>itr = vpi_iterate(Obj, ref_h); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>
	<p>A tagged <i>one-to-many</i> relationship is traversed similarly, using <i>Tag</i> instead of <i>Obj</i>:</p> <p>Example:</p> <pre>itr = vpi_iterate(Tag, ref_h); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>
	<p>A <i>one-to-many</i> relationship which originates from a circle is traversed using <code>NULL</code> for the <code>ref_h</code>:</p> <p>Example:</p> <pre>itr = vpi_iterate(Obj, NULL); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>

For relationships which do not have a tag, the type used for access is determined by adding `vpi` to the beginning of the word within the enclosure with each word's first letter being a capital. Refer to 26.3 for more details on VPI access constant names.

26.6 Object data model diagrams

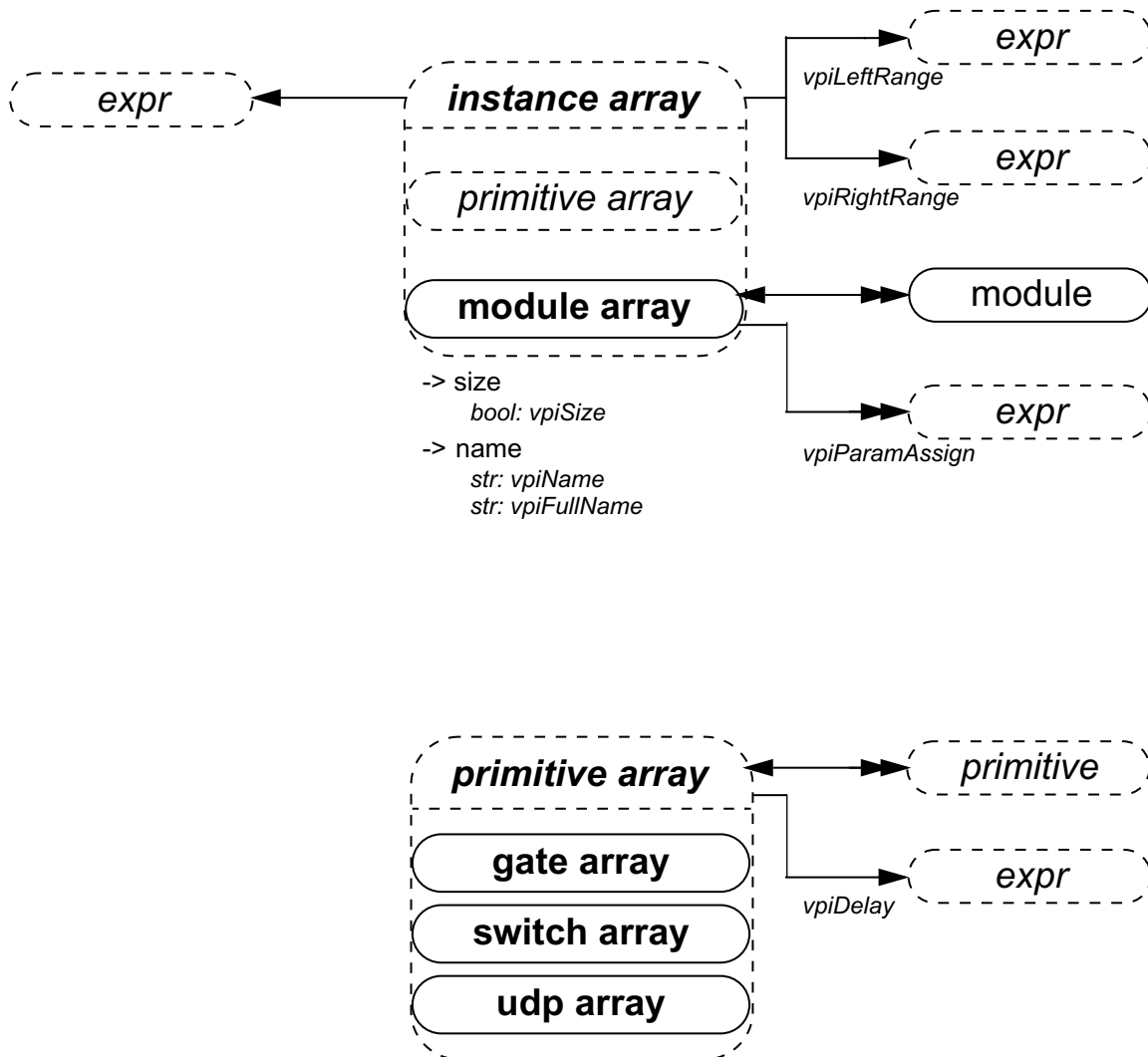
Subclauses 26.6.1 through 26.6.43 contain the data model diagrams that define the accessible objects and groups of objects, along with their relationships and properties.

26.6.1 Module



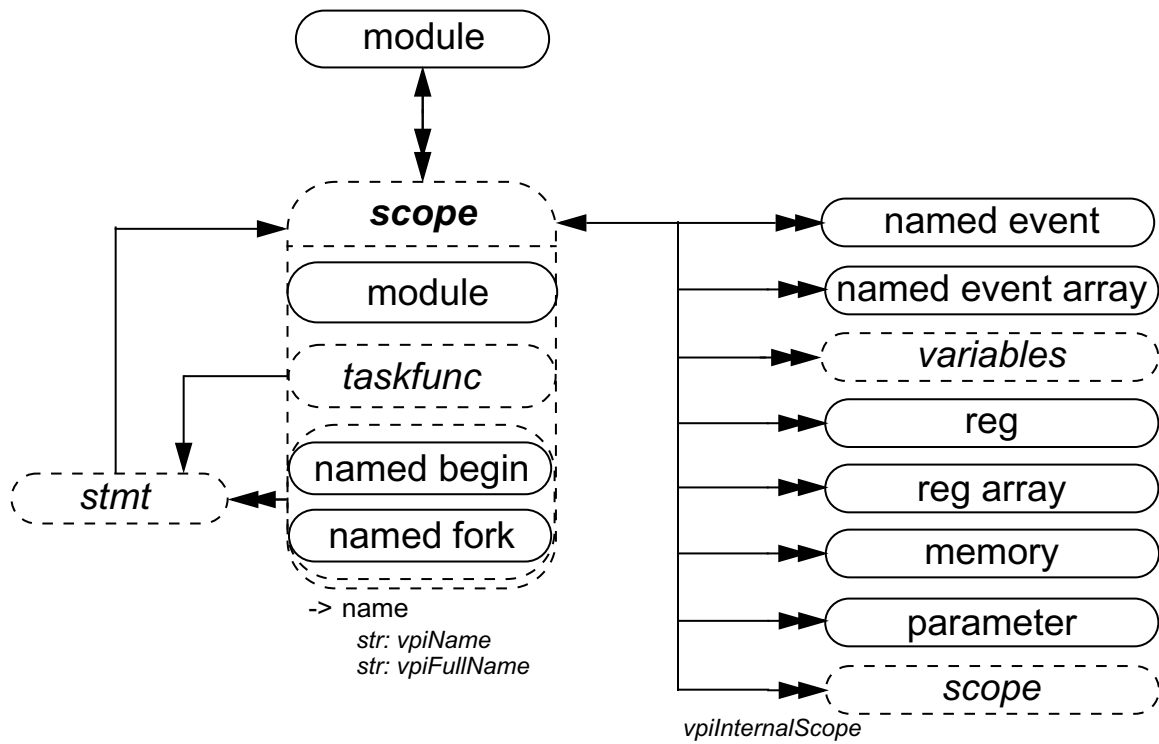
NOTES

- 1 Top-level modules shall be accessed using **vpi_iterate()** with a NULL reference object.
- 2 Passing a NULL handle to **vpi_get()** with types **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.
- 3 The properties **vpiDefLineNo** and **vpiDefFile** can be affected by the **`line** and **`file** compiler directives. See Clause 19 for more details on these compiler directives.
- 4 If a module is an element within a module array, the **vpiIndex** transition is used to access the index within the array. If a module is not part of a module array, this transition shall return NULL.

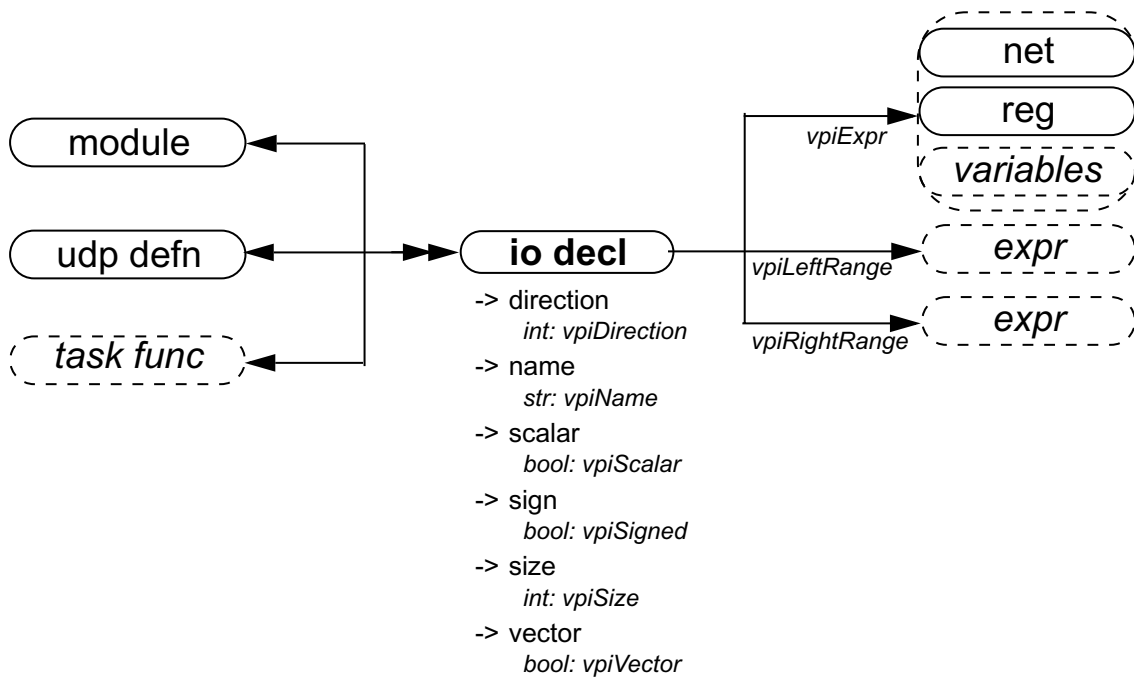
26.6.2 Instance arrays

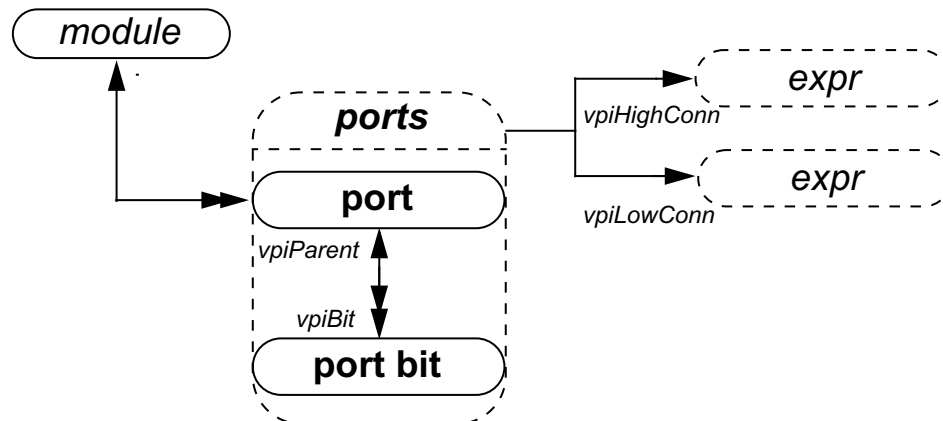
NOTE Traversing from the instance array to **expr** shall return a simple expression object of type **vpiOperation** with a **vpiOpType** of **vpiListOp**. This expression can be used to access the actual list of connections to the module or primitive instance array in the Verilog source code.

26.6.3 Scope



26.6.4 IO declaration



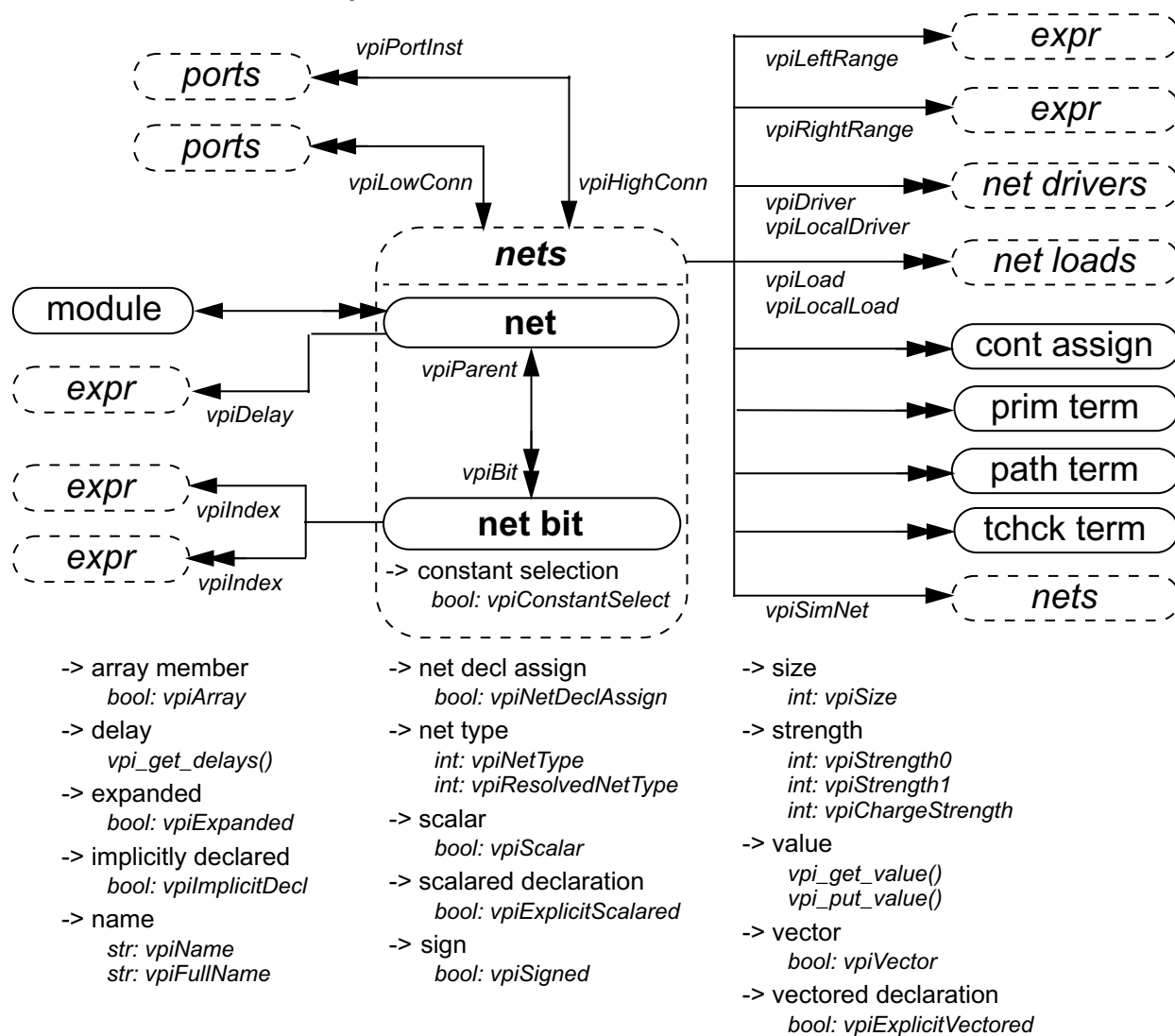
26.6.5 Ports

- > connected by name
bool: vpiConnByName
- > delay (mipd)
vpi_get_delays()
vpi_put_delays()
- > direction
int: vpiDirection
- > explicitly named
bool: vpiExplicitName
- > index
int: vpiPortIndex
- > name
str: vpiName
- > scalar
bool: vpiScalar
- > size
int: vpiSize
- > vector
bool: vpiVector

NOTES

- 1 **vpiHighConn** shall indicate the hierarchically higher (closer to the top module) port connection.
- 2 **vpiLowConn** shall indicate the lower (further from the top module) port connection.
- 3 Properties *scalar* and *vector* shall indicate if the port is 1 bit or more than 1 bit. They shall not indicate anything about what is connected to the port.
- 4 Properties *index* and *name* shall not apply for port bits.
- 5 If a port is explicitly named, then the explicit name shall be returned. If not, and a name exists, then that name shall be returned. Otherwise, **NULL** shall be returned.
- 6 **vpiPortIndex** can be used to determine the port order. The first port has a port index of zero.
- 7 **vpiHighConn** and **vpiLowConn** shall return **NULL** if the port is not connected.
- 8 **vpiSize** for a null port shall return 0.

26.6.6 Nets and net arrays



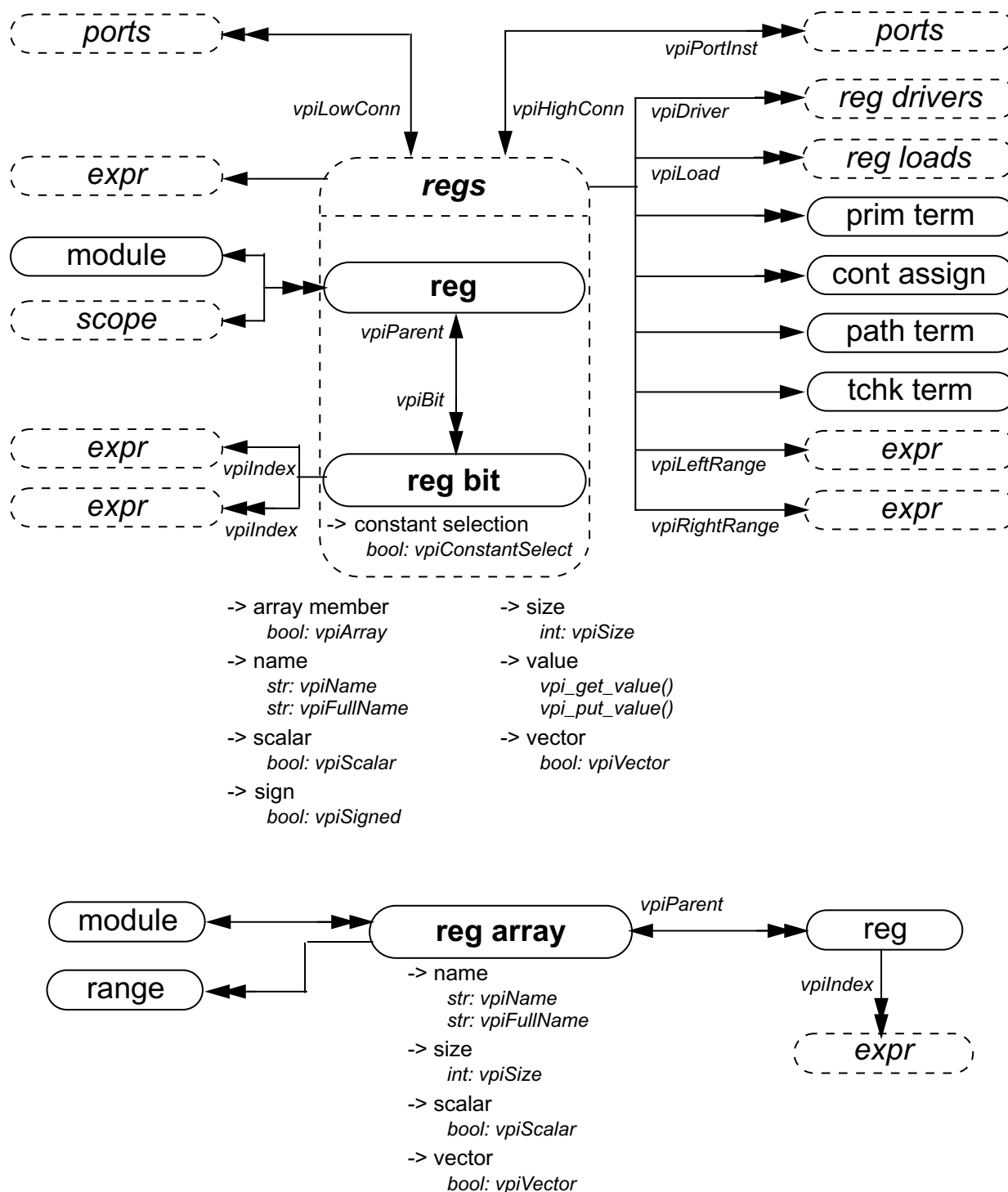
NOTES

1 For vectors, net bits shall be available regardless of vector expansion.

(Notes continued on next page)

- 2 Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
- 3 Continuous assignments and primitive terminals shall only be accessed from scalar nets or bit selects.
- 4 For **vpiPorts**, if the reference handle is a bit then port bits shall be returned. If it is the entire vector, then a handle to the entire port shall be returned.
- 5 For **vpiPortInst**, if the reference handle is a bit or scalar, then port bits or scalar ports shall be returned, unless the highconn for the port is a complex expression where the bit index cannot be determined. If this is the case, then the entire port shall be returned. If the reference handle is a vector, then the entire port shall be returned.
- 6 For **vpiPortInst**, it is possible for the reference handle to be part of the highconn expression, but not connected to any of the bits of the port. This may occur if there is a size mismatch. In this situation, the port shall not qualify as a member for that iteration.
- 7 For implicit nets, **vpiLineNo** shall return 0, and **vpiFile** shall return the file name where the implicit net is first referenced.
- 8 **vpi_handle(vpiIndex, net_bit_handle)** shall return the bit index for the net bit. **vpi_iterate(vpiIndex, net_bit_handle)** shall return the set of indices for a multidimensional net array bit select, starting with the index for the net bit and working outward.
- 9 Only active forces and assign statements shall be returned for **vpiLoad**.
- 10 Only active forces shall be returned for **vpiDriver**.
- 11 **vpiDriver** shall also return ports that are driven by objects other than nets and net bits.
- 12 **vpiLocalLoad** and **vpiLocalDriver** return only the loads or drivers that are local, i.e.: contained by the module instance which contains the net, including any ports connected to the net (output and inout ports are loads, input and inout ports are drivers).
- 13 For **vpiLoad**, **vpiLocalLoad**, **vpiDriver** and **vpiLocalDriver** iterators, if the object is **vpiNet** for a vector net, then all loads or drivers are returned exactly once as the loading or driving object. That is, if a part select loads or drives only some bits, the load or driver returned is the part select. If a driver is repeated, it is only returned once. To trace exact bit by bit connectivity pass a **vpiNetBit** object to **vpi_iterate**.
- 14 An iteration on loads or drivers for a variable bit-select shall return the set of loads or drivers for whatever bit that the bit-select is referring to at the beginning of the iteration.
- 15 **vpiSimNet** shall return a unique net if an implementation collapses nets across hierarchy (refer to Section 12.3.10 for the definition of simulated net and collapsed net).
- 16 The property **vpiExpanded** on an object of type **vpiNetBit** shall return the property's value for the parent.
- 17 The loads and drivers returned from **vpi_iterate(vpiLoad, obj_handle)** and **vpi_iterate(vpiDriver, obj_handle)** may not be the same in different implementations, due to allowable net collapsing (see Section 12.3.10). The loads and drivers returned from **vpi_iterate(vpiLocalLoad, obj_handle)** and **vpi_iterate(vpiLocalDriver, obj_handle)** shall be the same for all implementations.
- 18 The boolean property **vpiConstantSelect** returns TRUE if the expression that constitutes the index or indices evaluates to a constant, and FALSE otherwise.
- 19 **vpi_get(vpiSize, net_handle)** returns the number of bits in the net. **vpi_get(vpiSize, net_array_handle)** returns the total number of nets in the in the array.
- 20 **vpi_iterate(vpiIndex, net_handle)** shall return the set of indices for a net within an array, starting with the index for the net and working outward. If the net is not part of an array, a NULL shall be returned.

26.6.7 Regs and reg arrays



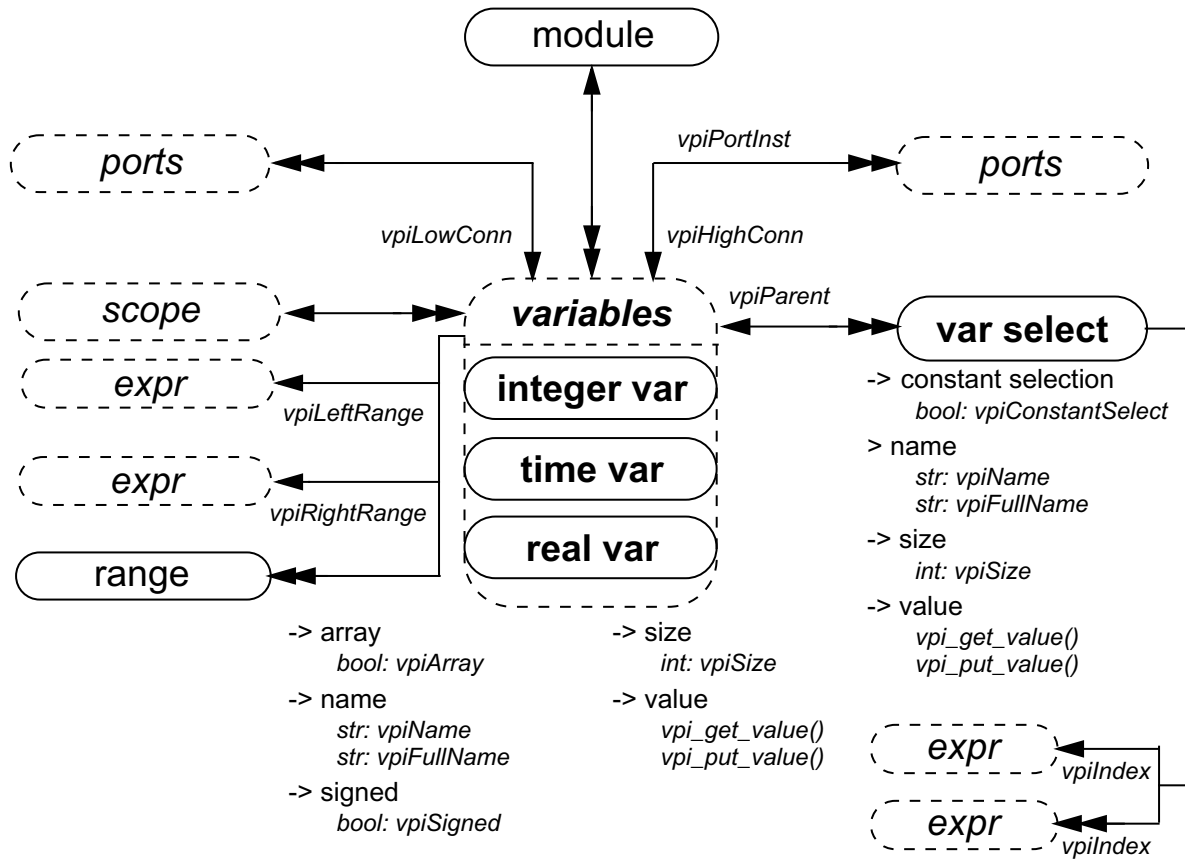
NOTES

- 1 Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
- 2 Continuous assignments and primitive terminals shall only be accessed from scalar regs and bit selects.

(Notes continued on next page)

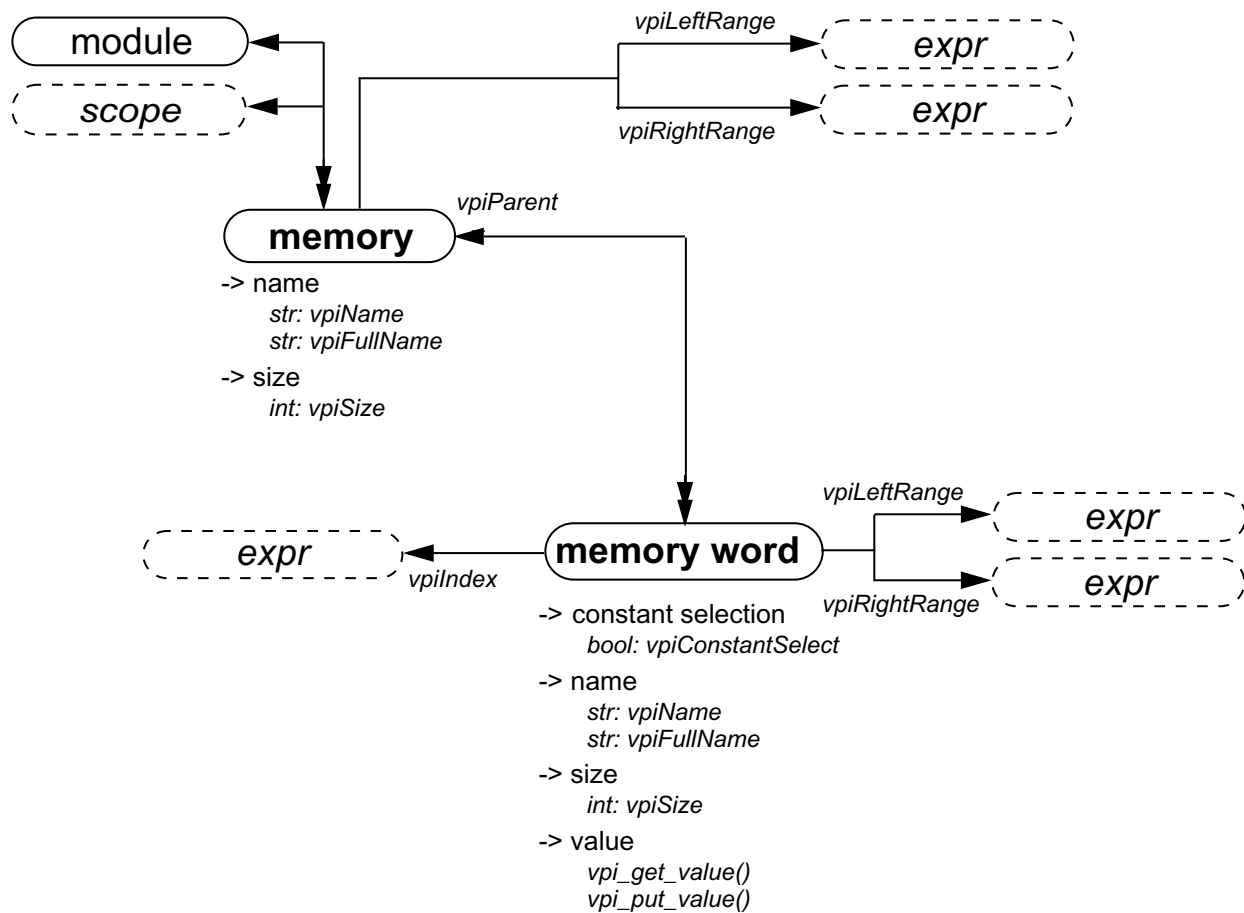
- 3 For **vpiPorts**, if the reference handle is a bit then port bits shall be returned. If it is the entire vector, then a handle to the entire port shall be returned.
- 4 For **vpiPortInst**, if the reference handle is a bit or scalar, then port bits or scalar ports shall be returned, unless the highconn for the port is a complex expression where the bit index cannot be determined. If this is the case, then the entire port shall be returned. If the reference handle is a vector, then the entire port shall be returned.
- 5 For **vpiPortInst**, it is possible for the reference handle to be part of the highconn expression, but not connected to any of the bits of the port. This may occur if there is a size mismatch. In this case, the port shall not qualify as a member for that iteration.
- 6 **vpi_handle(vpiIndex, reg_bit_handle)** shall return the bit index for the reg bit. **vpi_iterate(vpiIndex, reg_bit_handle)** shall return the set of indices for a multidimensional reg array bit select, starting with the index for the reg bit and working outward.
- 7 Only active forces and assign statements shall be returned for **vpiLoad** and **vpiDriver**.
- 8 For **vpiLoad** and **vpiDriver** iterators, if the object is **vpiReg** for a vectored reg, then all loads or drivers are returned exactly once as the loading or driving object. That is, if a part select loads or drives only some bits, the load or driver returned is the part select. If a driver is repeated, it is only returned once. To trace exact bit by bit connectivity, pass a **vpiRegBit** object to the iterator.
- 9 The loads and drivers returned from **vpi_iterate(vpiLoad, obj_handle)** and **vpi_iterate(vpiDriver, obj_handle)** may not be the same in different implementations due to allowable net collapsing (see Section 12.3.10).
- 10 An iteration on loads or drivers for a variable bit-select shall return the set of loads or drivers for whatever bit that the bit-select is referring to at the beginning of the iteration.
- 11 If the reg has a default initialization assignment, the expression can be accessed using **vpi_handle(vpiExpr, reg_handle)** or **vpi_handle(vpiExpr, reg_bit_handle)**.
- 12 **vpi_get(vpiSize, reg_handle)** returns the number of bits in the reg. **vpi_get(vpiSize, reg_array_handle)** returns the total number of regs in the in the array.
- 13 **vpi_iterate(vpiIndex, reg_handle)** shall return the set of indices for a reg within an array, starting with the index for the reg and working outward. If the reg is not part of an array, a NULL shall be returned.

26.6.8 Variables



NOTES

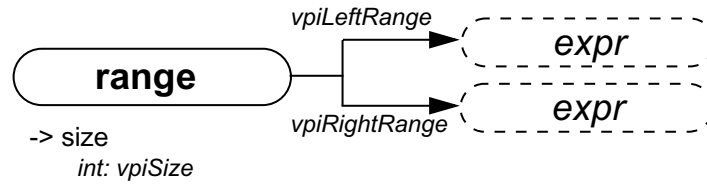
- 1 A var select is a word selected from a variable array.
- 2 The VPI does not provide access to bits of variables. If a handle to bit select of a variable is obtained, the object shall be a **vpiBitSelect** in the simple expression class. The variable containing the bit can be accessed using **vpiParent**. Refer to Section 26.6.25.
- 3 The boolean property **vpiArray** shall be **TRUE** if the variable handle references an array of variables, and **FALSE** otherwise. If the variable is an array, iterate on **vpiVarSelect** to obtain handles to each variable in the array.
- 4 **vpi_handle(vpiIndex, var_select_handle)** shall return the index of a var select in a 1-dimensional array. **vpi_iterate(vpiIndex, var_select_handle)** shall return the set of indices for a var select in a multidimensional array, starting with the index for the var select and working outward.
- 5 **vpiLeftRange** and **vpiRightRange** shall apply to variables when **vpiArray** is **TRUE**, and represent the array range declaration. These relationships are only valid when **vpiArray** is **TRUE**.
- 6 **vpiSize** for a variable array shall return the number of variables in the array. For non-array variables, it shall return the size of the variable in bits.
- 7 **vpiSize** for a var select shall return the number of bits in the var select.
- 8 Variables whose boolean property **vpiArray** is **TRUE** do not have a value property.

26.6.9 Memory

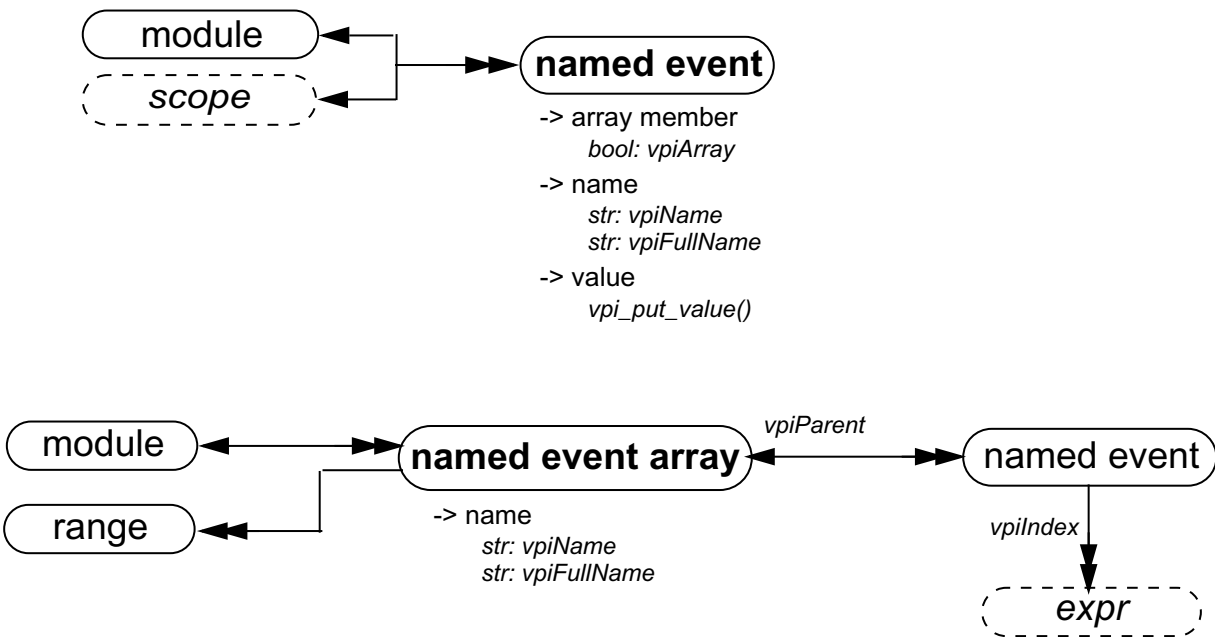
NOTES

- 1 **vpiSize** for a memory shall return the number of words in the memory.
- 2 **vpiSize** for a memory word shall return the number of bits in the word.
- 3 A memory is a one-dimensional array of reg types. Since 1364-2000 supports multi-dimensional arrays of regs, access to arrays of regs has been generalized. Although the access provided in Section 26.6.9 is still allowed, the preferred method is to iterate using **vpiRegArray**. See Section 26.6.7.

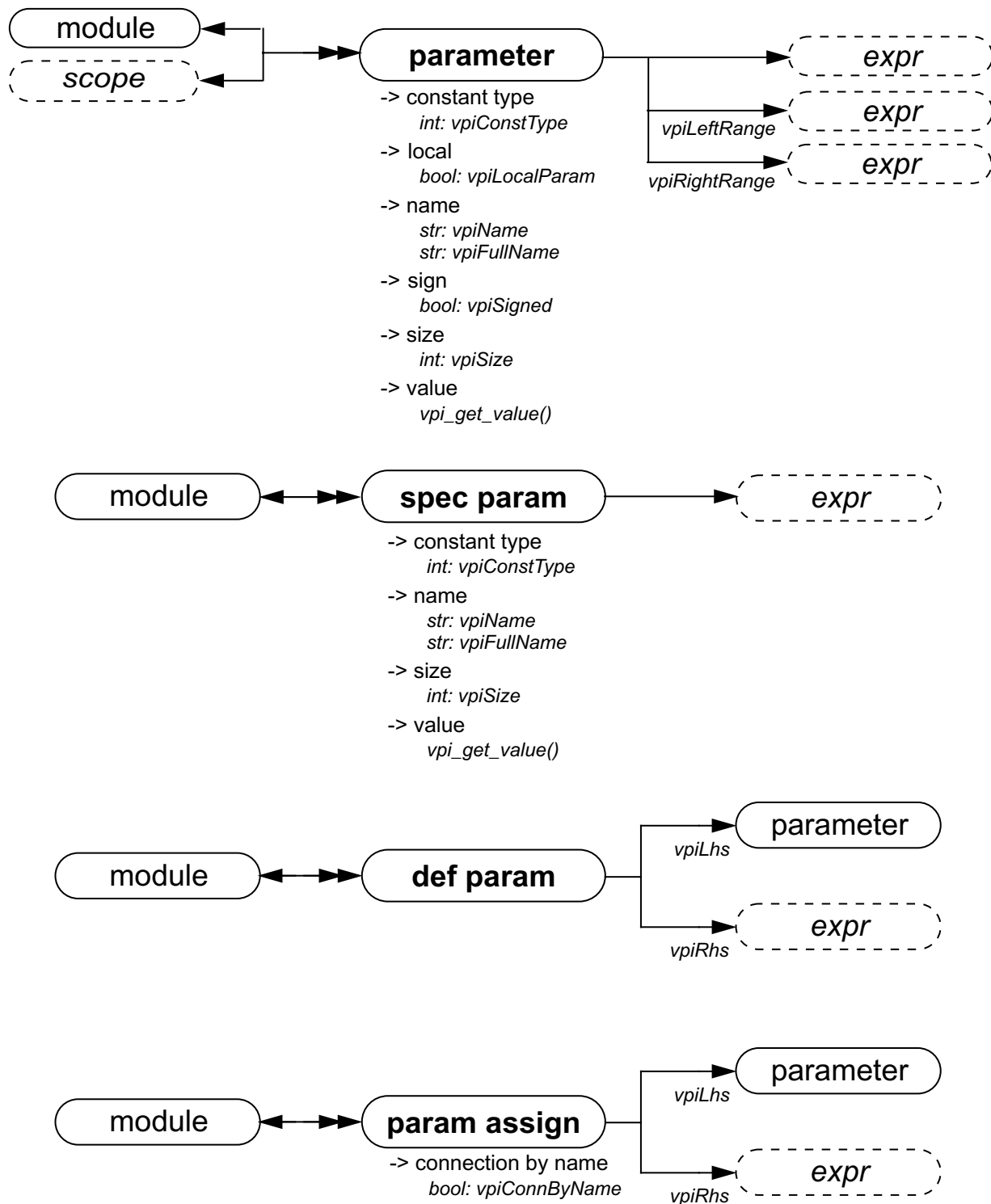
26.6.10 Object range



26.6.11 Named event



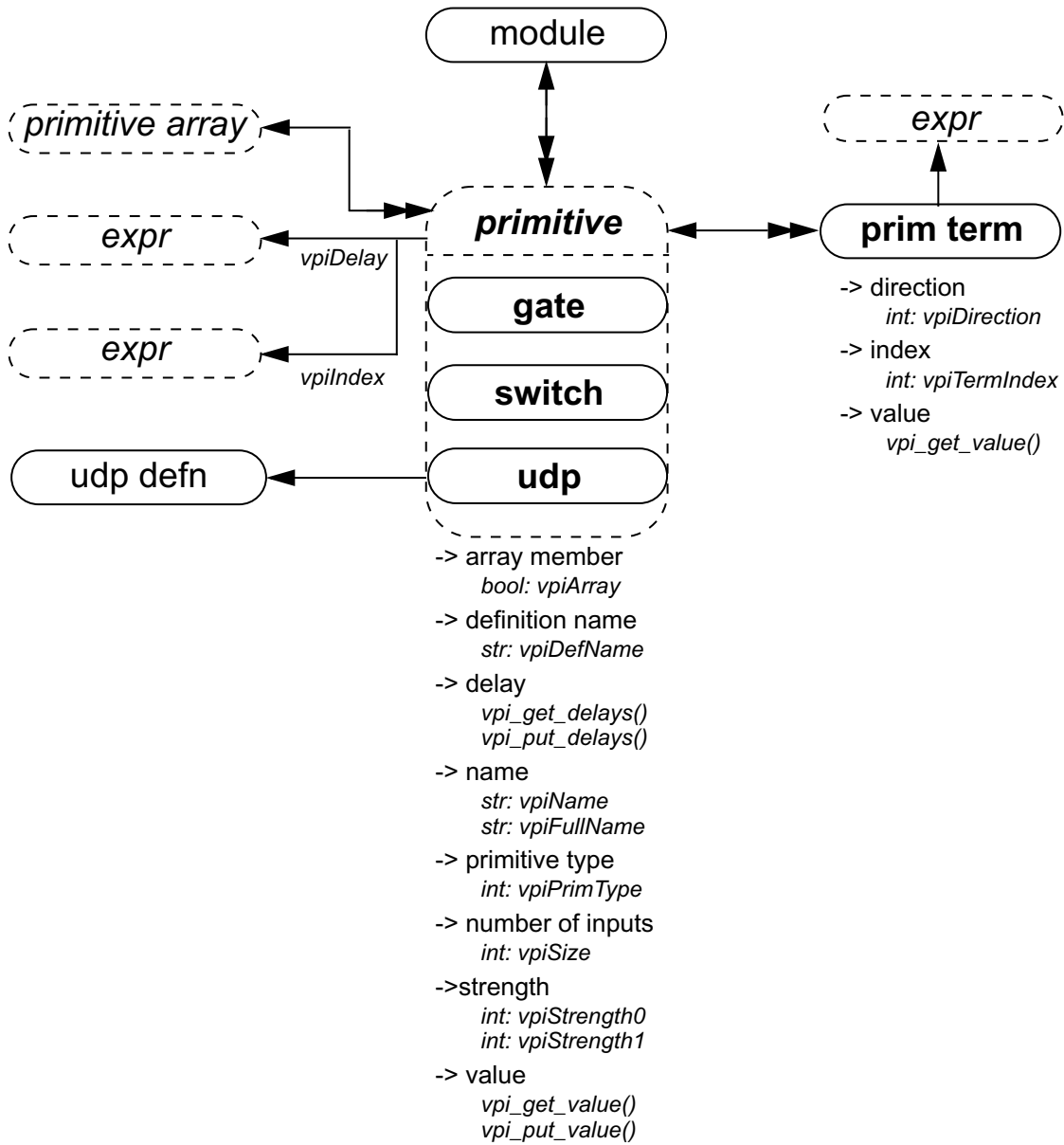
NOTE **vpi_iterate(vpiIndex, named_event_handle)** shall return the set of indices for a named event within an array, starting with the index for the named event and working outward. If the named event is not part of an array, a NULL shall be returned.

26.6.12 Parameter, specparam

NOTES

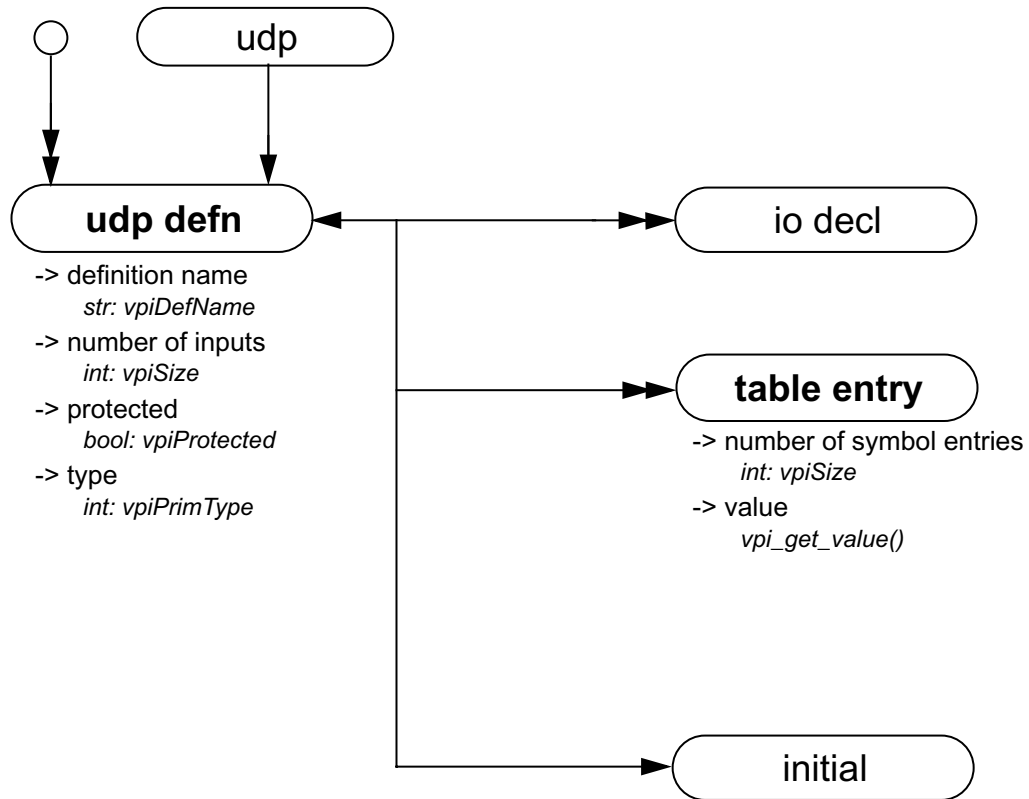
- 1 Obtaining the value from the object **parameter** shall return the final value of the parameter after all module instantiation overrides and defparams have been resolved.
- 2 **vpiLhs** from a param assign object shall return a handle to the overridden parameter.

26.6.13 Primitive, prim term



NOTES

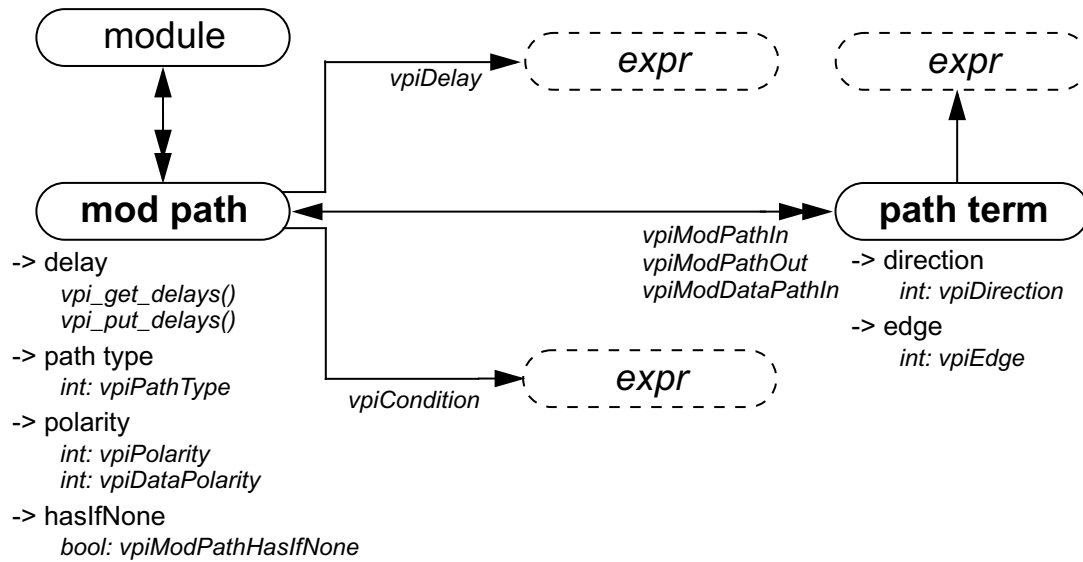
- 1 **vpiSize** shall return the number of inputs.
- 2 For primitives, **vpi_put_value()** shall only be used with sequential UDP primitives.
- 3 **vpiTermIndex** can be used to determine the terminal order. The first terminal has a term index of zero.
- 4 If a primitive is an element within a primitive array, the **vpiIndex** transition is used to access the index within the array. If a primitive is not part of a primitive array, this transition shall return NULL.

26.6.14 UDP

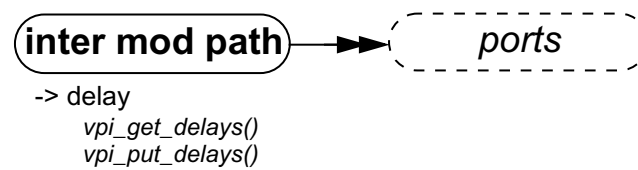
NOTES

- 1 Only string (decompilation) and vector (ASCII values) shall be obtained for table entry objects using **vpi_get_value()**. Refer to the definition of **vpi_get_value()** for additional details.
- 2 **vpiPrimType** returns **vpiSeqPrim** for sequential UDP's and **vpiCombPrim** for combinatorial UDP's.

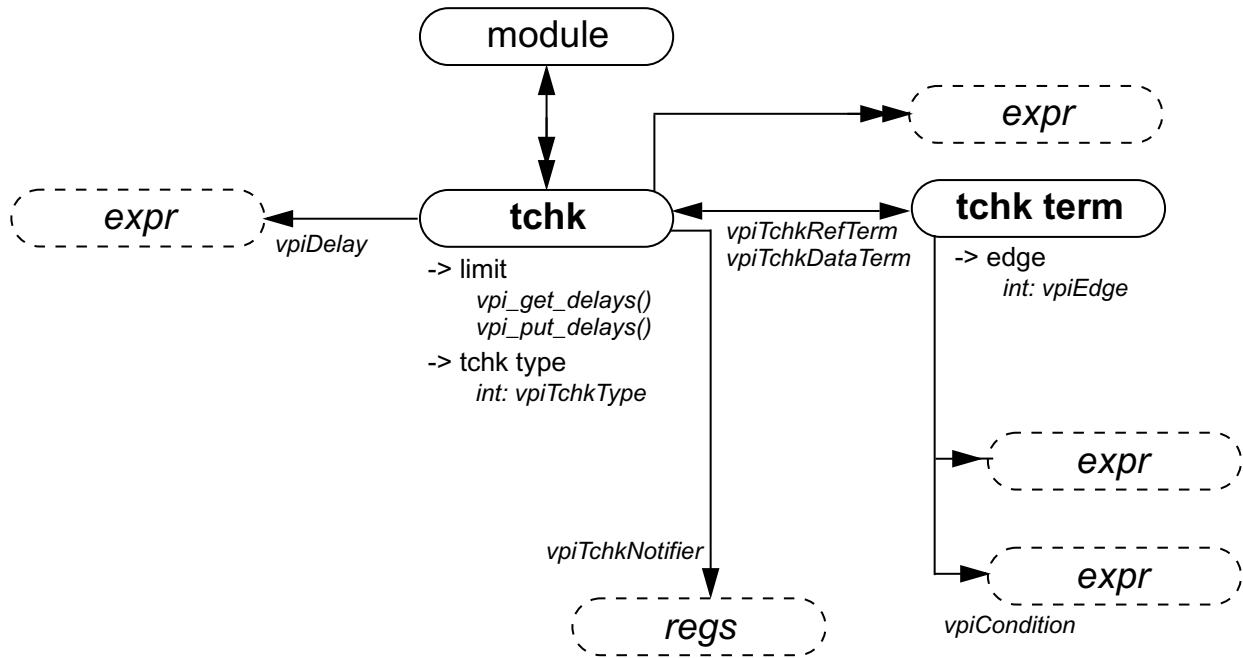
26.6.15 Module path, path term



26.6.16 Intermodule path

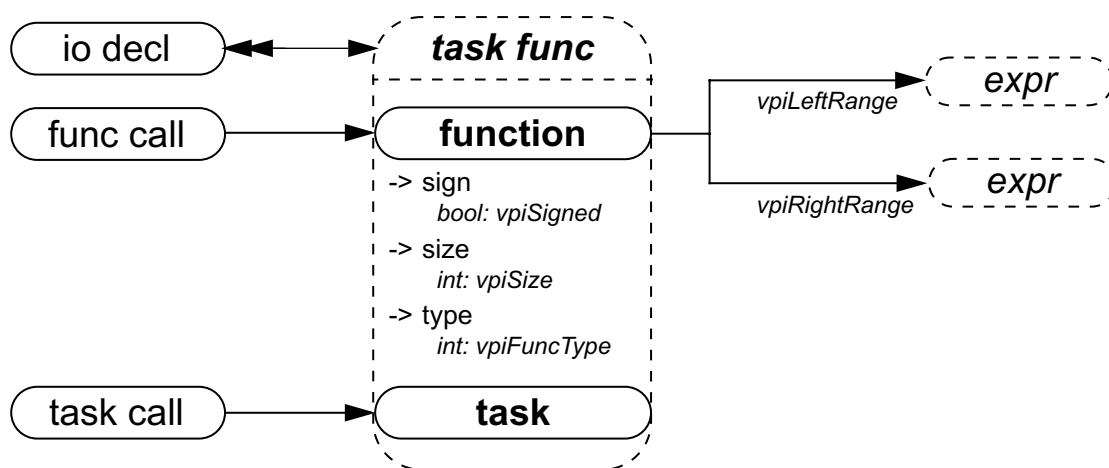


NOTE To get to an intermodule path, **vpi_handle_multi(vpiInterModPath, port1, port2)** can be used.

26.6.17 Timing check

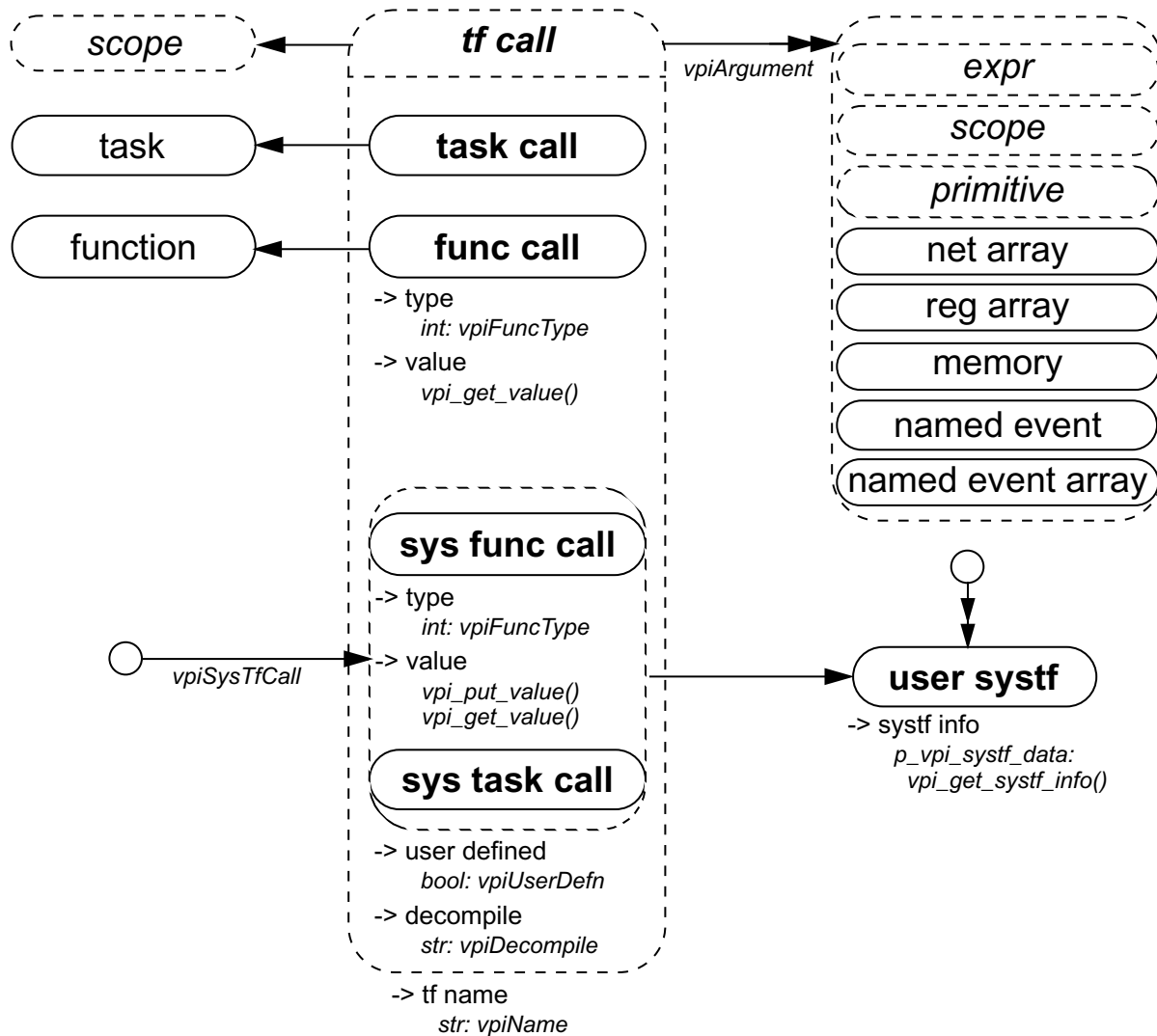
NOTES

- 1 The **vpiTchkRefTerm** is the first terminal for all tchks except **\$setup**, where **vpiTchkDataTerm** is the first terminal and **vpiTchkRefTerm** is the second terminal.
- 2 When iterating for the expressions in a check the handle returned for what is known as the data, ref, and notifier terminal will have the type **vpiTchkTerm**. All other arguments will have types matching the expression.

26.6.18 Task, function declaration

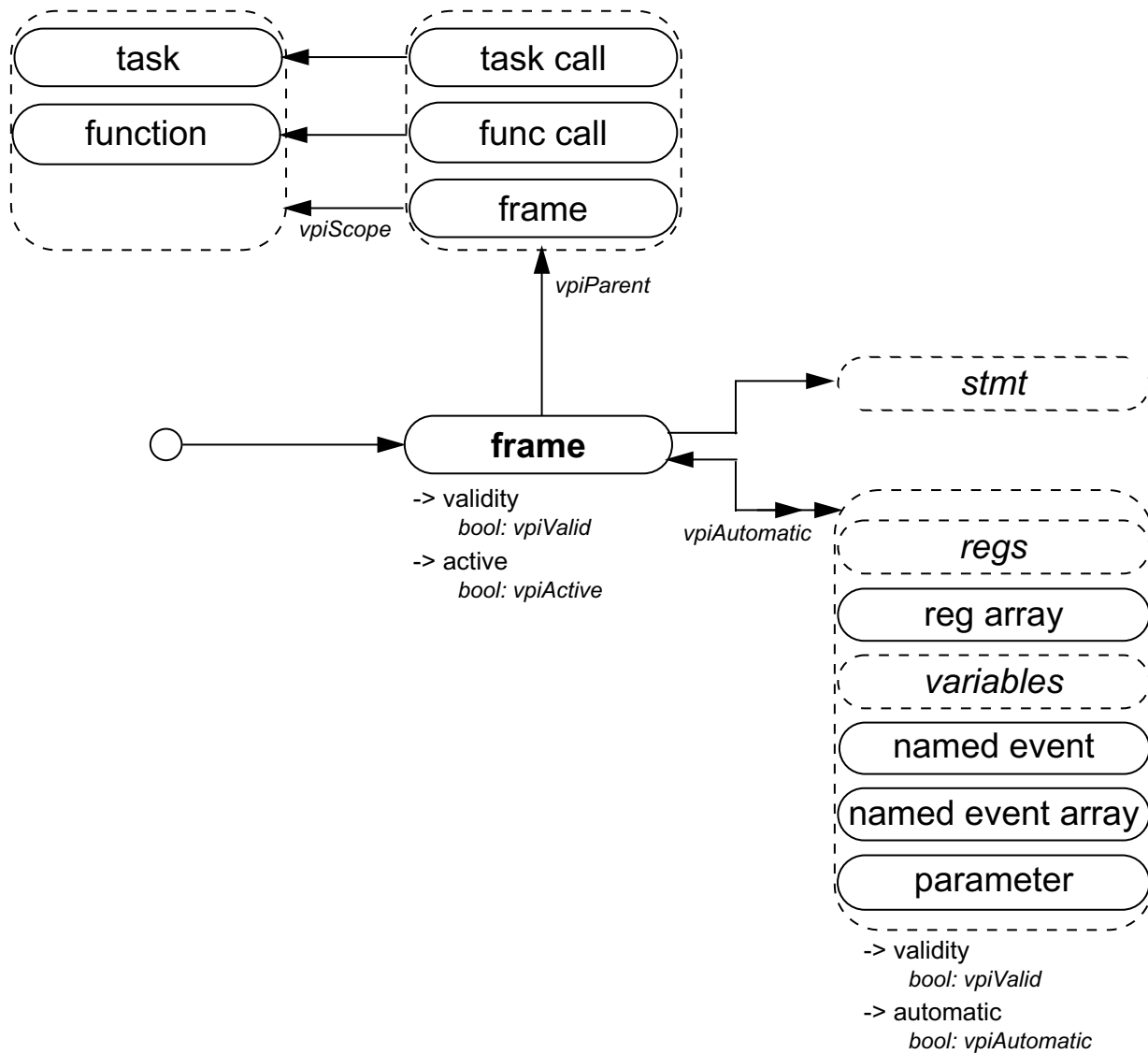
NOTE A Verilog HDL function shall contain an object with the same name, size, and type as the function.

26.6.19 Task and function call



NOTES

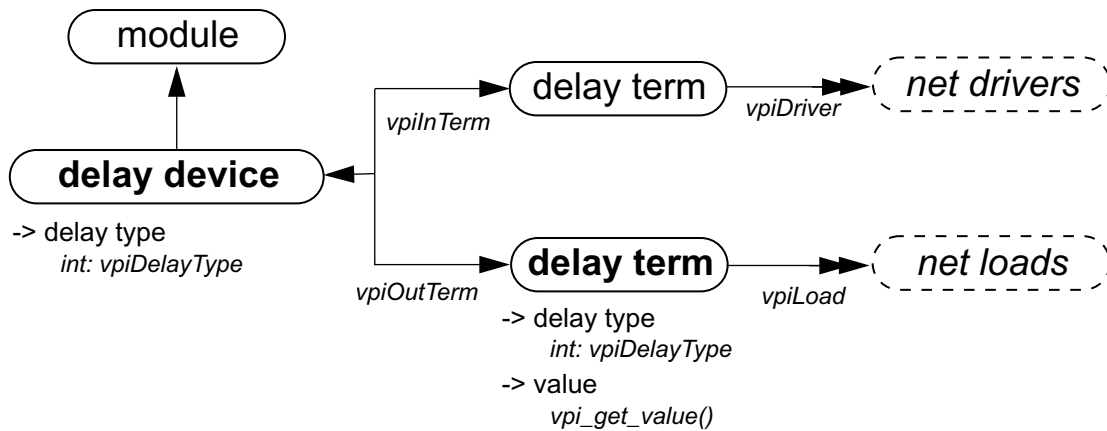
- 1 The system task or function that invoked an application shall be accessed with **vpi_handle(vpiSysTfCall, NULL)**
- 2 **vpi_get_value()** shall return the current value of the system function.
- 3 If the **vpiUserDefn** property of a system task or function call is true, then the properties of the corresponding systf object shall be obtained via **vpi_get_systf_info()**.
- 4 All user-defined system tasks or functions shall be retrieved using **vpi_iterate()**, with **vpiUserSystf** as the type argument, and a **NULL** reference argument.
- 5 Arguments to PLI tasks or functions are not evaluated until an application requests their value. Effectively, the value of any argument is not known until the application asks for it. When an argument is an HDL or system function call, the function cannot be evaluated until the application asks for its value. If the application never asks for the value of the function, it is never evaluated. If the application has a handle to an HDL or system function it may ask for its value at any time in the simulation. When this happens the function is called and evaluated at this time.
- 6 A null argument is an expression with a **vpiType** of **vpiOperation** and a **vpiOpType** of **vpiNullOp**.
- 7 The property **vpiDecompile** will return a string with a functionally equivalent system task or function call to what was in the original HDL. The arguments will be decompiled using the same manner as any expression is decompiled. See Section 26.6.26 for a description of expression decompilation.

26.6.20 Frames

NOTES

- 1 It shall be illegal to place value change callbacks on automatic variables.
- 2 It shall be illegal to put a value with a delay on automatic variables.
- 3 There is at most only one active frame at any time. To get a handle to the currently active frame, use **vpi_handle(vpiFrame, NULL)**. The **frame** to **stmt** transition shall return the currently active statement within the frame.
- 4 Frame handles must be freed using **vpi_free_object()** once the application no longer needs the handle. If the handle is not freed it shall continue to exist, even after the frame has completed execution.

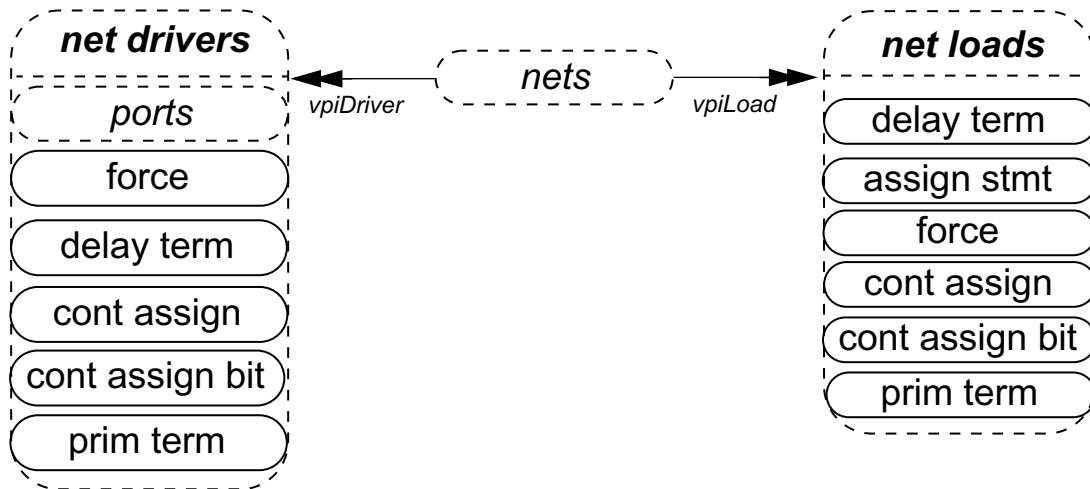
26.6.21 Delay terminals



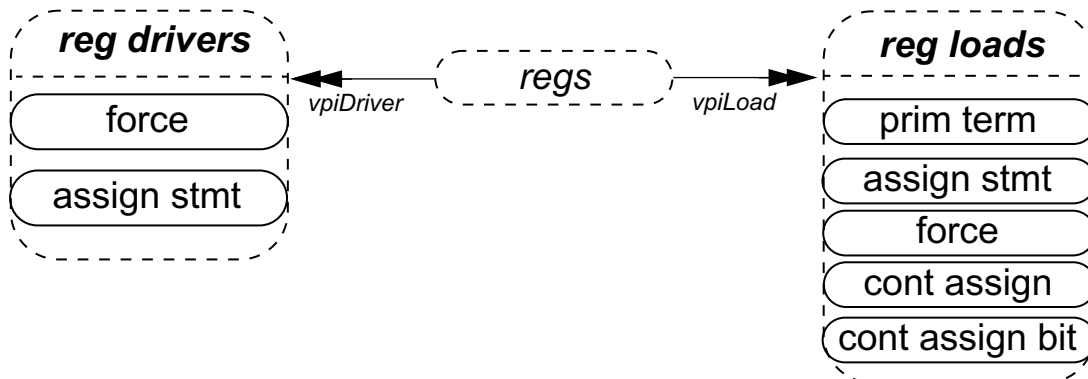
NOTES

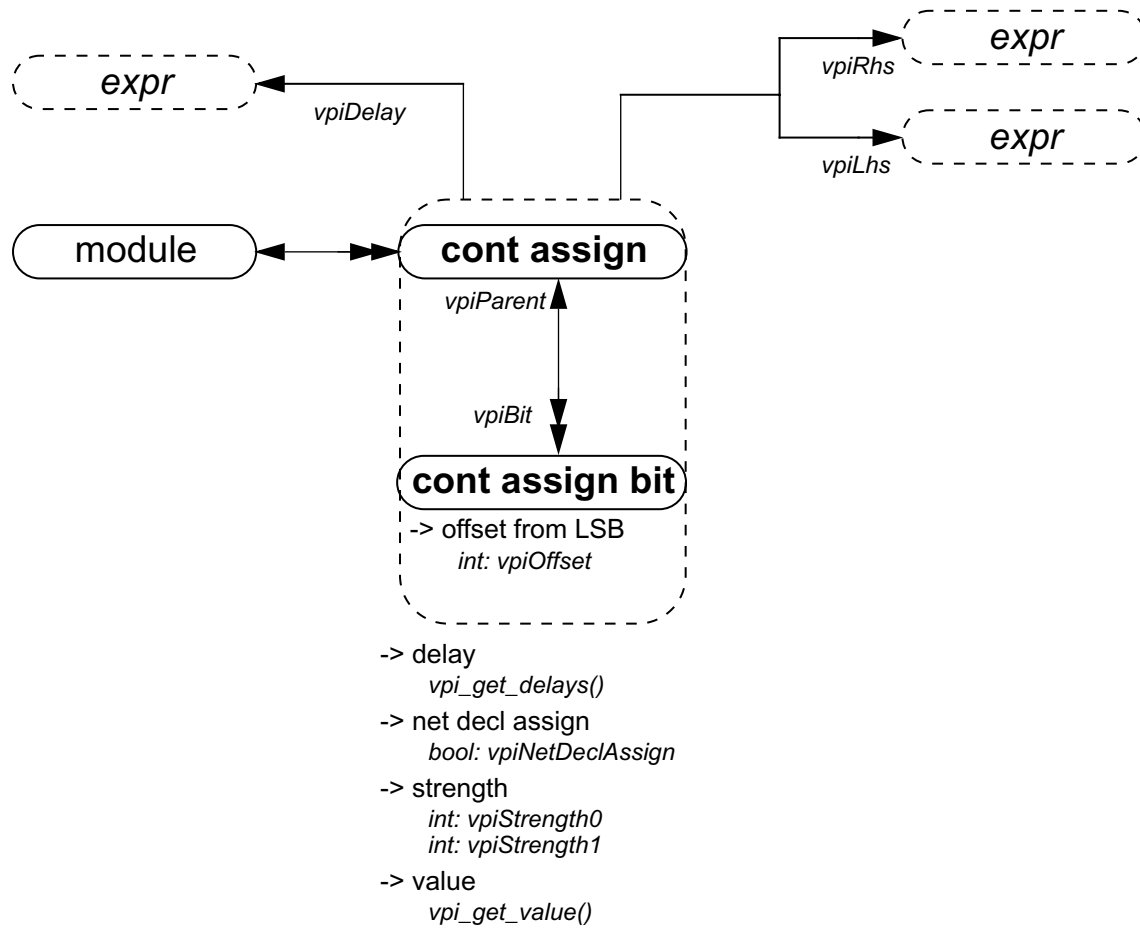
- 1 The value of the input delay term shall change before the delay associated with the delay device.
- 2 The value of the output delay term shall not change until after the delay has occurred.

26.6.22 Net drivers and loads



26.6.23 Reg drivers and loads

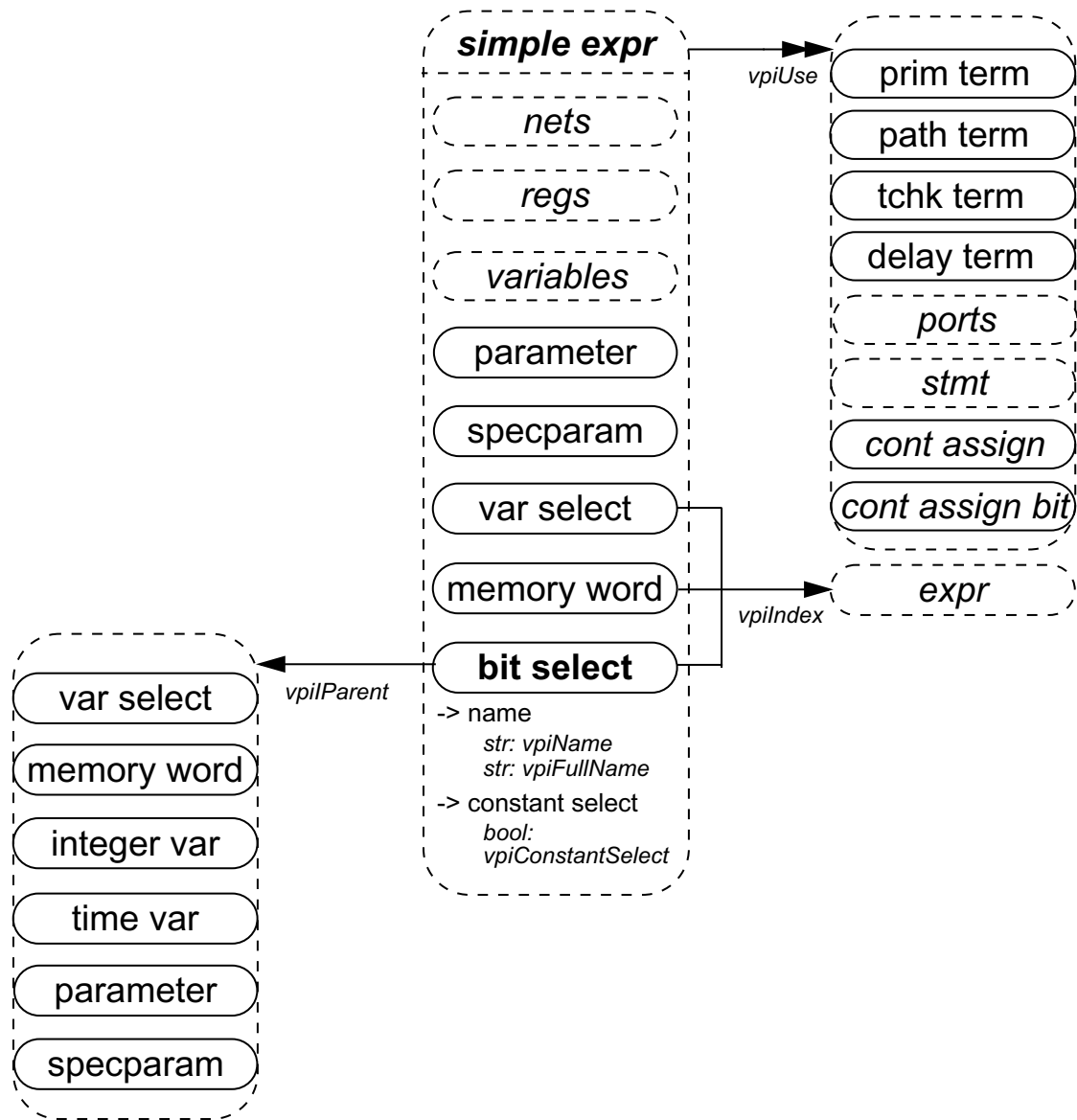


26.6.24 Continuous assignment

NOTES

- 1 The size of a cont assign bit is always scalar.
- 2 Callbacks for value changes can be placed onto cont assign or a cont assign bit.
- 3 **vpiOffset** shall return zero for the least significant bit.

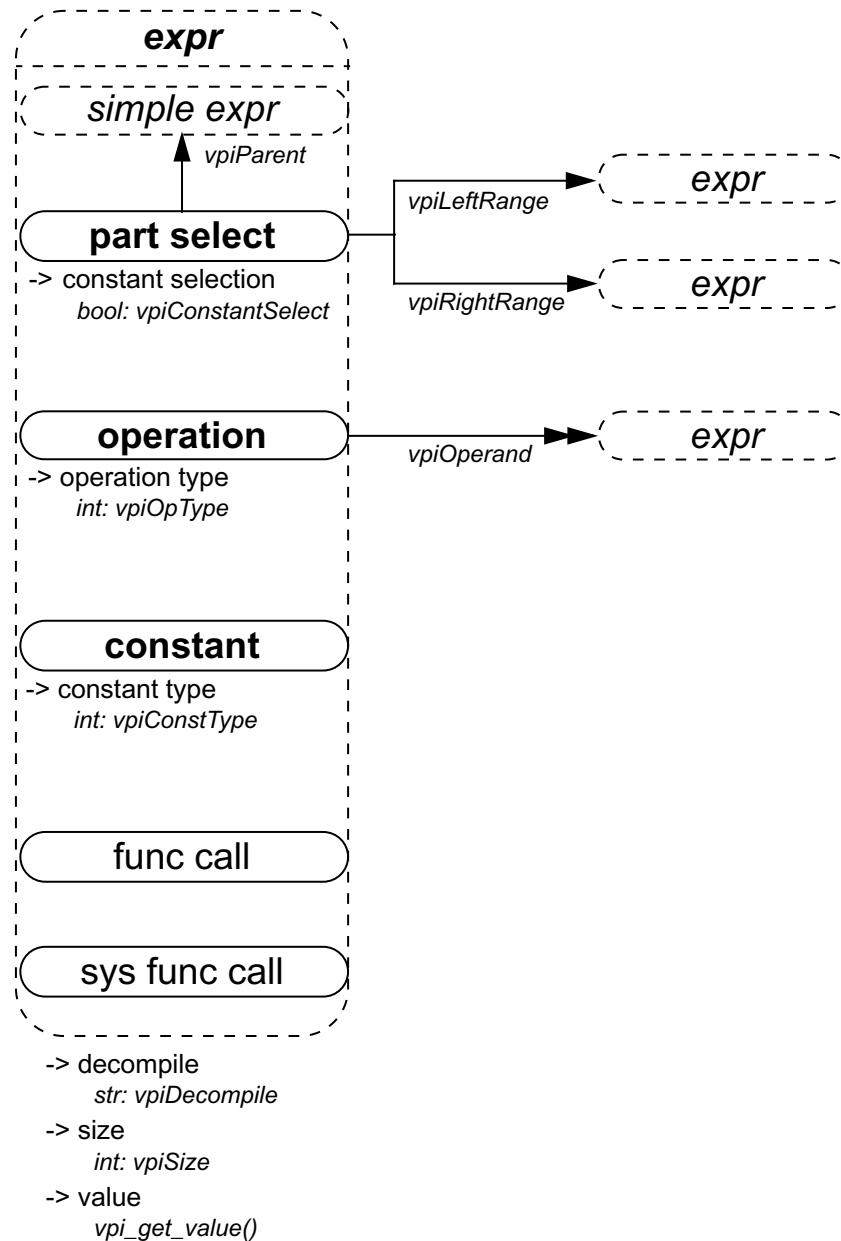
26.6.25 Simple expressions



NOTES

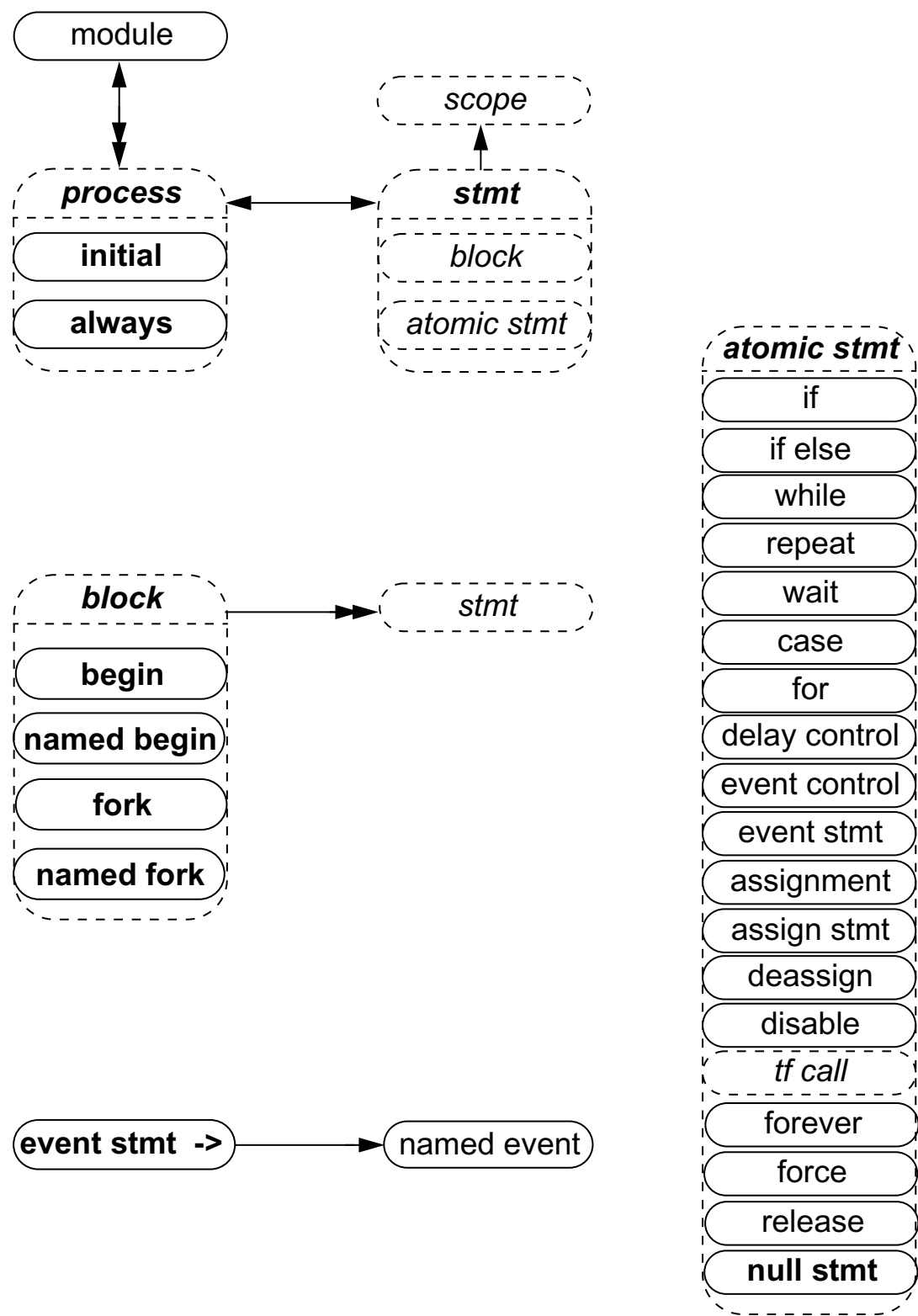
1 For vectors, the **vpiUse** relationship shall access any use of the vector or part-selects or bit-selects thereof.

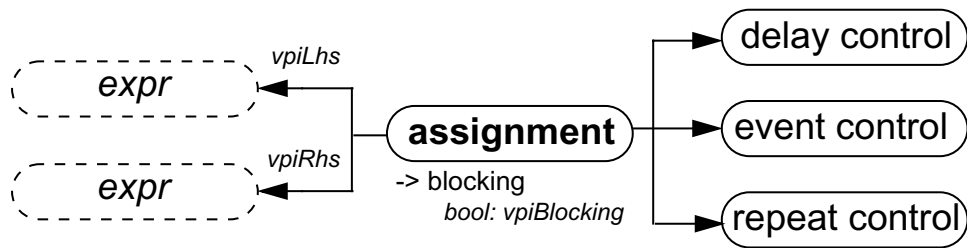
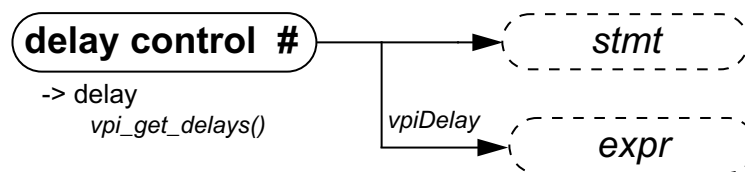
2 For bit-selects, the **vpiUse** relationship shall access any specific use of that bit, any use of the parent vector, and any part-select that contains that bit.

26.6.26 Expressions**NOTES**

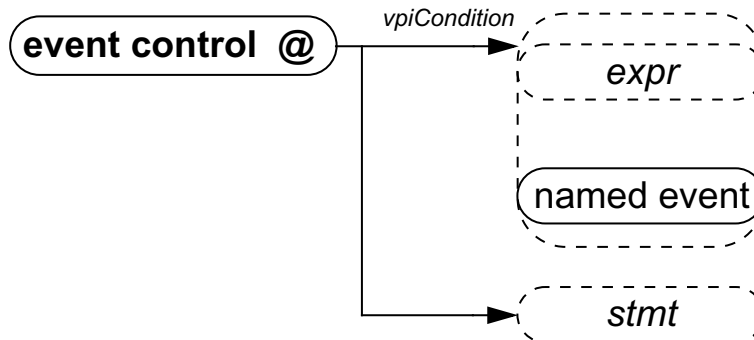
- 1 For an operator whose type is **vpiMultiConcat**, the first operand shall be the multiplier expression. The remaining operands shall be the expressions within the concatenation.
- 2 The property **vpiDecompile** will return a string with a functionally equivalent expression to the original expression within the HDL. Parenthesis shall be added only to preserve precedence. Each operand and operator shall be separated by a single space character. No additional white space shall be added due to parenthesis.

26.6.27 Process, block, statement, event statement

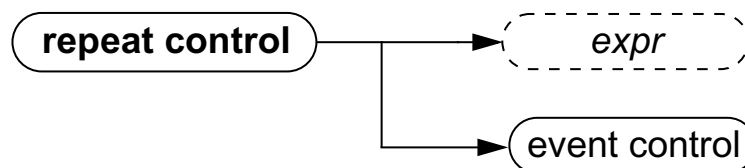


26.6.28 Assignment**26.6.29 Delay control**

NOTE For delay control associated with assignment, the statement shall always be NULL.

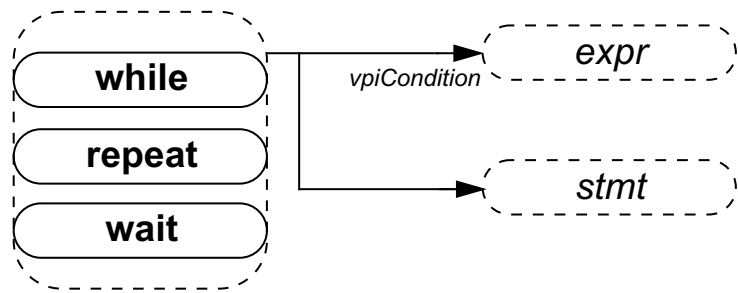
26.6.30 Event control

NOTE For event control associated with assignment, the statement shall always be NULL.

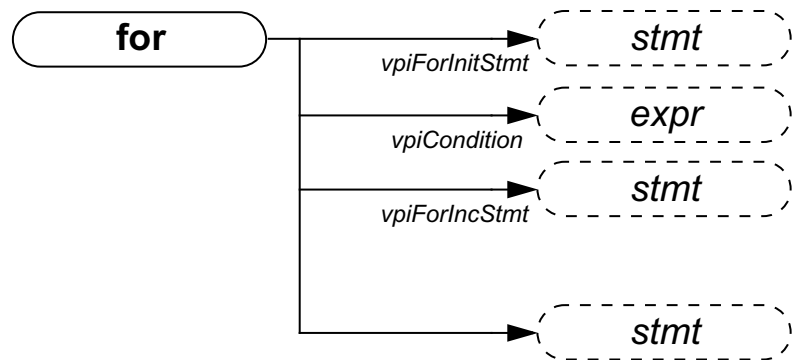
26.6.31 Repeat control

NOTE For delay control and event control associated with assignment, the statement shall always be NULL.

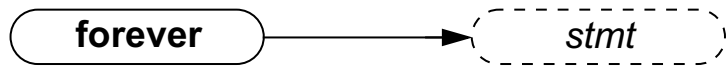
26.6.32 While, repeat, wait

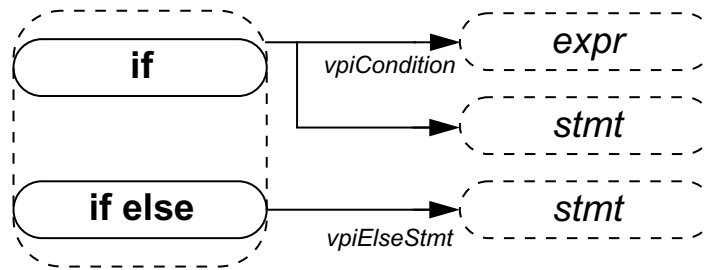
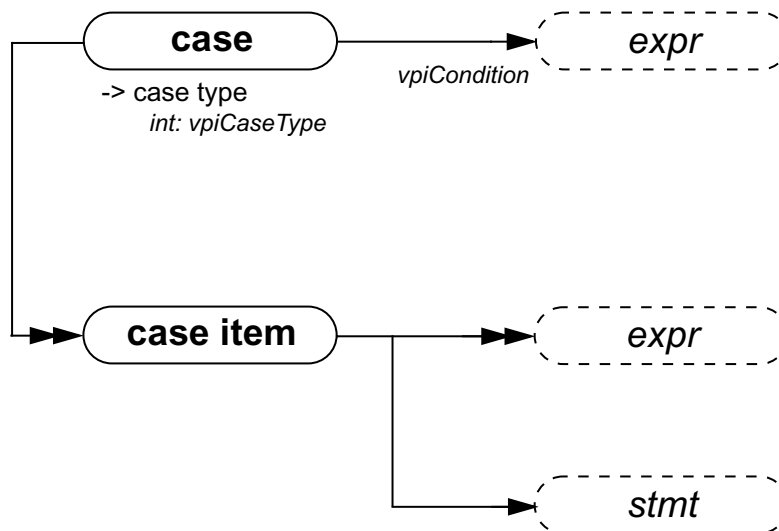


26.6.33 For



26.6.34 Forever

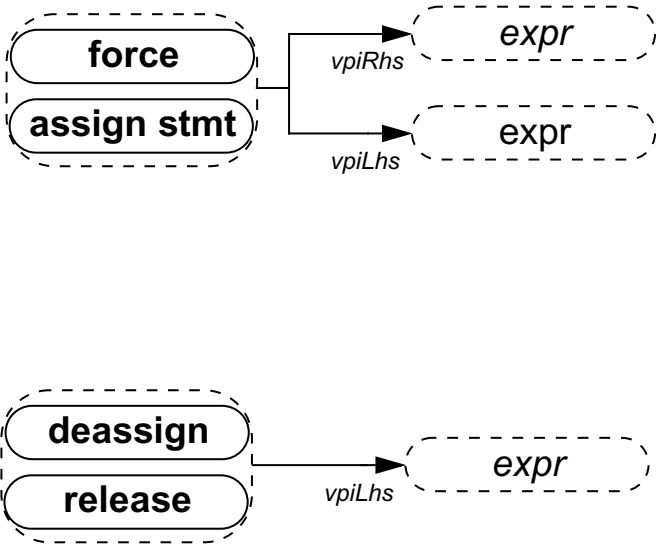


26.6.35 If, if-else**26.6.36 Case**

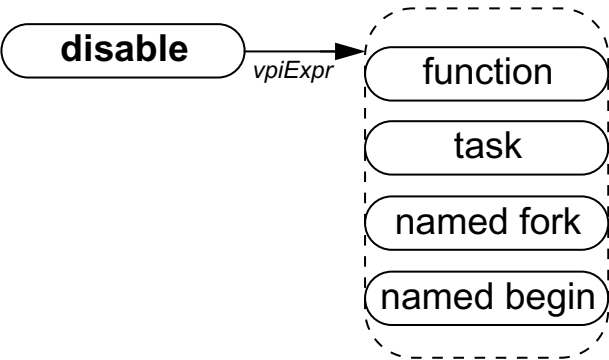
NOTES

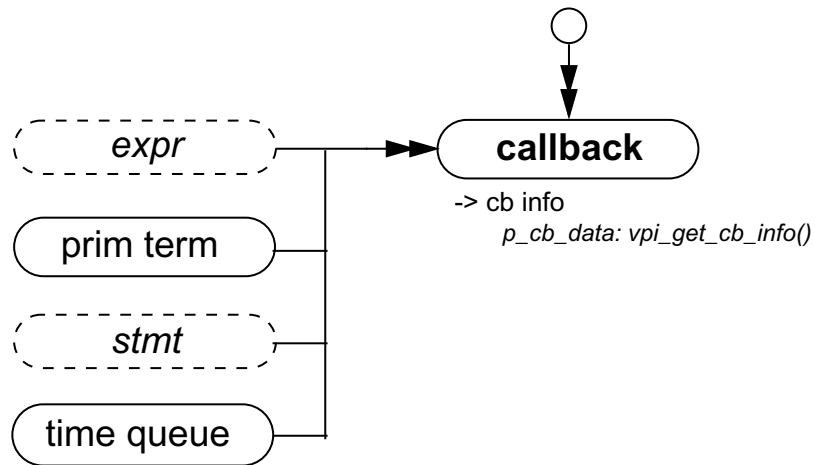
- 1 The *case item* shall group all case conditions that branch to the same statement.
- 2 **vpi_iterate()** shall return **NULL** for the default case item since there is no expression with the default case.

26.6.37 Assign statement, deassign, force, release



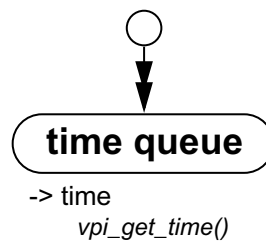
26.6.38 Disable



26.6.39 Callback

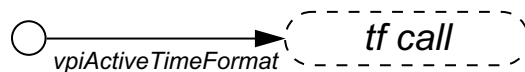
NOTES

- 1 To get information about the callback object, the routine **vpi_get_cb_info()** can be used.
- 2 To get callback objects not related to the above objects, the second argument to **vpi_iterate()** shall be NULL.

26.6.40 Time queue

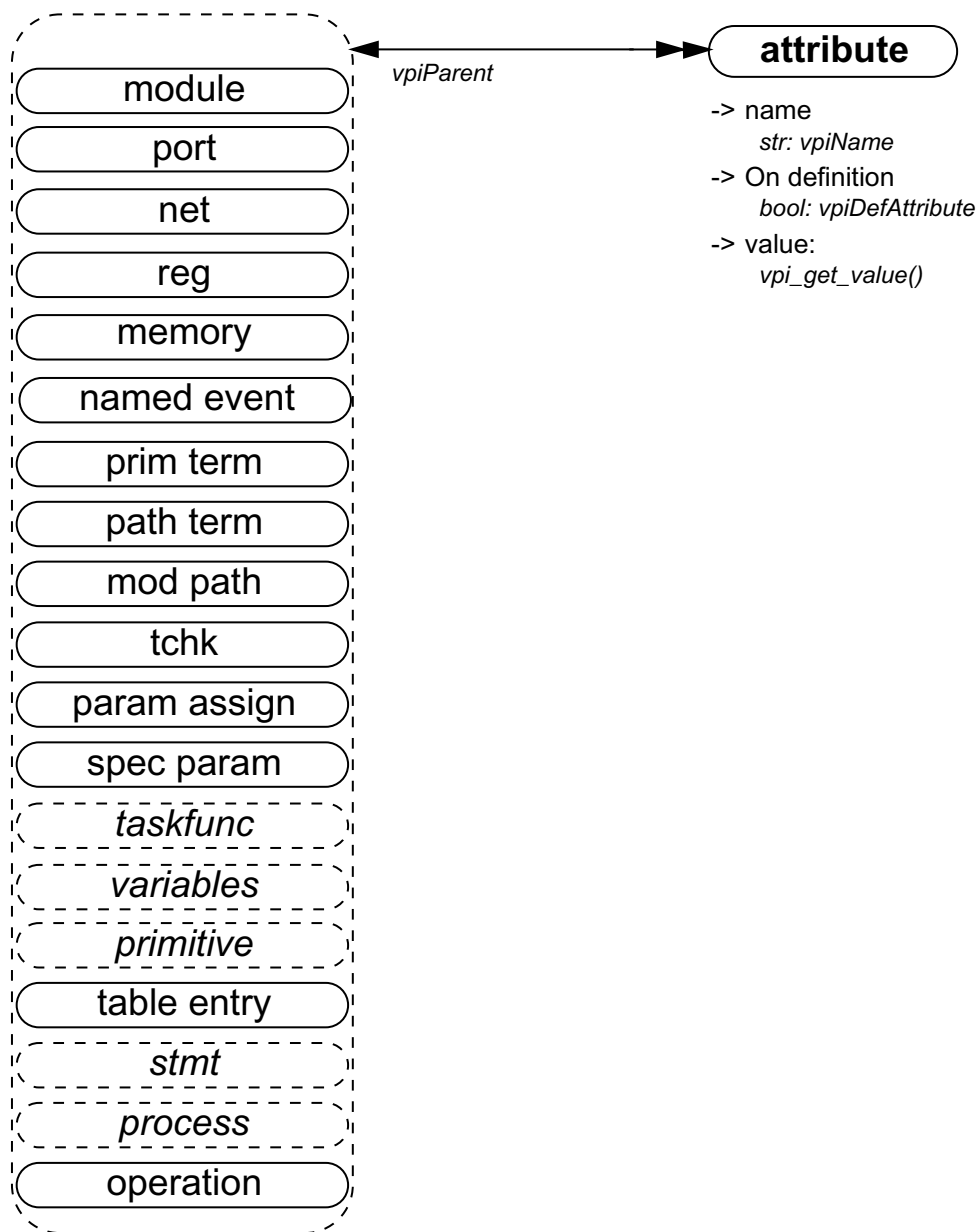
NOTES

- 1 The time queue objects shall be returned in increasing order of simulation time.
- 2 **vpi_iterate()** shall return NULL if there is nothing left in the simulation queue.
- 3 If any events after read only sync remain in the current queue, then it shall not be returned as part of the iteration.

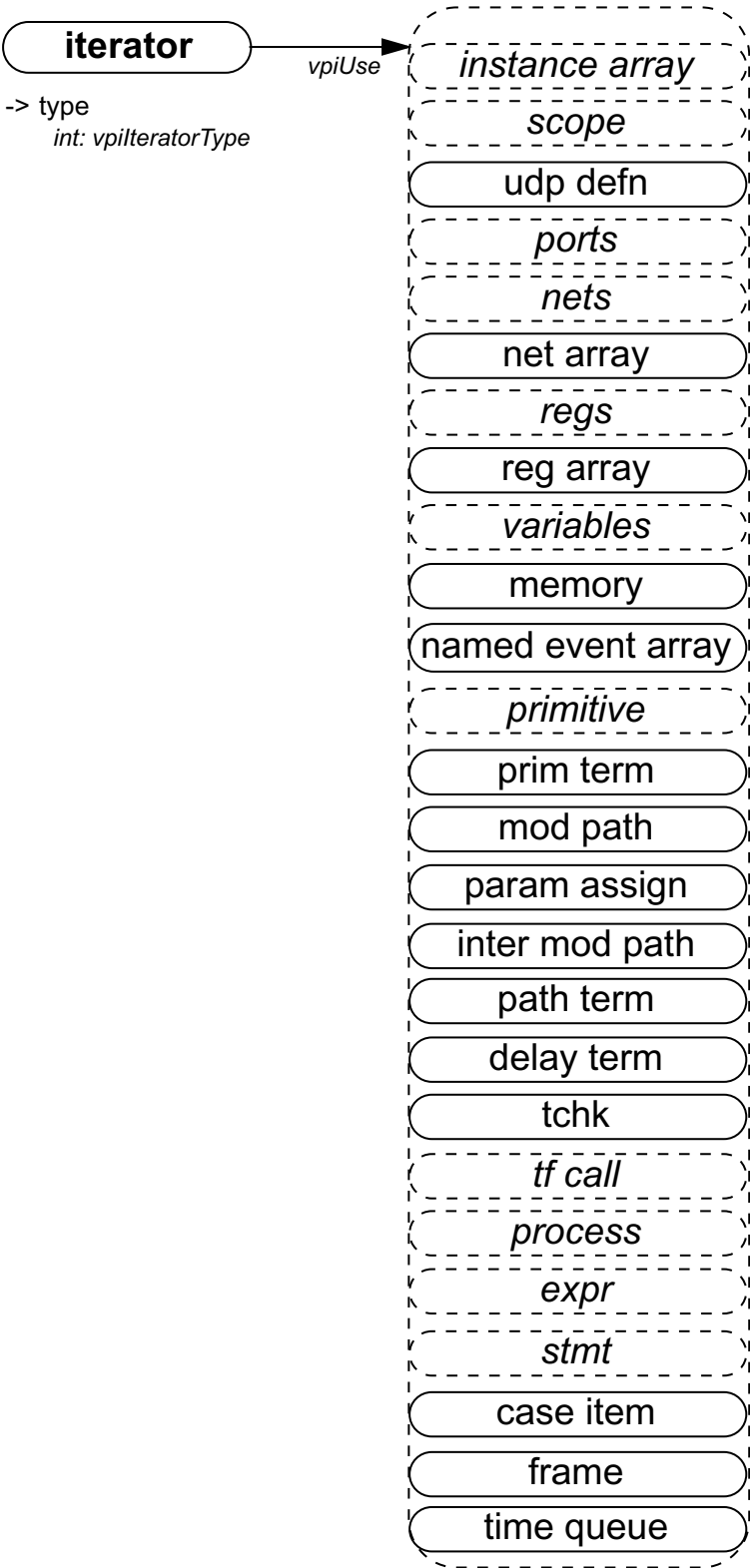
26.6.41 Active time format

NOTE If `$timeformat()` has not been called, `vpi_handle(vpiActiveTimeFormat, NULL)` shall return a NULL.

26.6.42 Attributes



26.6.43 Iterator



STU28

NOTES

- 1 **vpi_handle(vpiUse, iterator_handle)** shall return the reference handle used to create the iterator.
- 2 It is possible to have a `NULL` reference handle, in which case **vpi_handle(vpiUse, iterator_handle)** shall return `NULL`.

27. VPI routine definitions

This clause describes the Verilog Procedural Interface (VPI) routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order. See Clause 23 for the conventions used in the definitions of the PLI routines.

27.1 vpi_chk_error()

vpi_chk_error()			
Synopsis:	Retrieve information about VPI routine errors.		
Syntax:	vpi_chk_error(error_info_p)		
Returns:	Type	Description	
	PLI_INT32	returns the error severity level if the previous VPI routine call resulted in an error and 0 (false) if no error occurred	
Arguments:	Type	Name	Description
	p_vpi_error_info	error_info_p	Pointer to a structure containing error information

The VPI routine **vpi_chk_error()** shall return an integer constant representing an error severity level if the previous call to a VPI routine resulted in an error. The error constants are shown in Table 212. If the previous call to a VPI routine did not result in an error, then **vpi_chk_error()** shall return **0** (false). The error status shall be reset by any VPI routine call except **vpi_chk_error()**. Calling **vpi_chk_error()** shall have no effect on the error status.

Table 212—Return error constants for vpi_chk_error()

Error Constant	Severity Level
vpiNotice	lowest severity ↓ highest severity
vpiWarning	
vpiError	
vpiSystem	
vpiInternal	

If an error occurred, the `s_vpi_error_info` structure shall contain information about the error. If the error information is not needed, a `NULL` can be passed to the routine. The `s_vpi_error_info` structure used by **vpi_chk_error()** is defined in `vpi_user.h` and is listed in Figure 172.

```

typedef struct t_vpi_error_info

    PLI_INT32 state;           /* vpi[ Compile,PLI,Run] */
    PLI_INT32 level;          /* vpi[ Notice,Warning,Error,System,Internal] */
    PLI_BYTE8 *message;
    PLI_BYTE8 *product;
    PLI_BYTE8 *code;
    PLI_BYTE8 *file;
    PLI_INT32 line;
    s_vpi_error_info, *p_vpi_error_info;

```

Figure 172—The s_vpi_error_info structure definition**27.2 vpi_compare_objects()**

vpi_compare_objects()			
Synopsis:	Compare two handles to determine if they reference the same object.		
Syntax:	vpi_compare_objects(obj1, obj2)		
		Type	Description
Returns:	PLI_INT32	1 (true) if the two handles refer to the same object. Otherwise, 0 (false)	
		Type	Name
		Description	
Arguments:	vpiHandle	obj1	Handle to an object
	vpiHandle	obj2	Handle to an object

The VPI routine **vpi_compare_objects()** shall return 1 (true) if the two handles refer to the same object. Otherwise, 0 (false) shall be returned. Handle equivalence cannot be determined with a C == comparison.

27.3 vpi_control()

vpi_control()			
Synopsis:	Pass information from user code to simulator.		
Syntax:	vpi_control(operation, varargs)		
		Type	Description
Returns:	PLI_INT32	1 (true) if successful; 0 (false) on a failure	
		Type	Name
		Description	
Arguments:	PLI_INT32	operation	select type of operation
		varargs	variable number of operation specific arguments
Related routines:			

The VPI routine `vpi_control()` shall pass information from a user PLI application to a Verilog software tool, such as a simulator. The following control constants are defined as part of the VPI standard:

vpiStop	causes the \$stop built-in Verilog system task to be executed upon return of the user function. This operation shall be passed one additional integer argument, which is the same as the diagnostic message level argument passed to \$stop (see 17.4.2).
vpiFinish	causes the \$finish built-in Verilog system task to be executed upon return of the user function. This operation shall be passed one additional integer argument, which is the same as the diagnostic message level argument passed to \$finish (see 17.4.1).
vpiReset	causes the \$reset built-in Verilog system task to be executed upon return of the user function. This operation shall be passed three additional integer arguments: <code>stop_value</code> , <code>reset_value</code> and <code>diagnostic_level</code> , which are the same values passed to the \$reset system task (see C.7).
vpiSetInteractiveScope	causes a tool's interactive scope to be immediately changed to a new scope. This operation shall be passed one additional argument, which is a <code>vpiHandle</code> object within the vpiScope class.

27.4 vpi_flush()

vpi_flush()		
Synopsis:	Flushes the data from the simulator output channel and log file output buffers.	
Syntax:	<code>vpi_flush()</code>	
	Type	Description
Returns:	PLI_INT32	0 if successful, non-zero if unsuccessful
	Type	Name
Arguments:	None	
Related routines:	Use <code>vpi_printf()</code> to write a finite number of arguments to the simulator output channel and log file Use <code>vpi_vprintf()</code> to write a variable number of arguments to the simulator output channel and log file Use <code>vpi_mcd_printf()</code> to write one or more opened files	

The routine **vpi_flush()** shall flush the output buffers for the simulator's output channel and current log file.

27.5 vpi_free_object()

vpi_free_object()		
Synopsis:	Free memory allocated by VPI routines.	
Syntax:	<code>vpi_free_object(obj)</code>	
	Type	Description
Returns:	PLI_INT32	1 (true) on success and 0 (false) on failure
	Type	Name
Arguments:	vpiHandle	obj
		Handle of an object

The VPI routine **vpi_free_object()** shall free memory allocated for objects. It shall generally be used to free memory created for iterator objects. The iterator object shall automatically be freed when **vpi_scan()** returns **NULL** either because it has completed an object traversal or encountered an error condition. If neither of these conditions occur (which can happen if the code breaks out of an iteration loop before it has scanned every object), **vpi_free_object()** should be called to free any memory allocated for the iterator. This routine can also optionally be used for implementations that have to allocate memory for objects. The routine shall return **1** (true) on success and **0** (false) on failure.

27.6 vpi_get()

vpi_get()			
Synopsis:	Get the value of an integer or boolean property of an object.		
Syntax:	vpi_get(prop, obj)		
		Type	Description
Returns:	PLI_INT32	Value of an integer or boolean property	
		Type	Name
Arguments:	PLI_INT32	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get_str() to get string properties		

The VPI routine **vpi_get()** shall return the value of integer and boolean object properties. These properties shall be of type **PLI_INT32**. Boolean properties shall have a value of **1** for TRUE and **0** for FALSE. For integer object properties such as **vpiSize**, any integer shall be returned. For integer object properties that return a defined value, refer to Annex G for the value that shall be returned. Note for object property **vpiTimeUnit** or **vpiTimePrecision**, if the object is **NULL**, then the simulation time unit shall be returned. Should an error occur, **vpi_get()** shall return **vpiUndefined**.

27.7 vpi_get_cb_info()

vpi_get_cb_info()			
Synopsis:	Retrieve information about a simulation-related callback.		
Syntax:	vpi_get_cb_info(obj, cb_data_p)		
		Type	Description
Returns:	void		
		Type	Name
Arguments:	vpiHandle	obj	Handle to a simulation-related callback
	p_cb_data	cb_data_p	Pointer to a structure containing callback information
Related routines:	Use vpi_get_systf_info() to retrieve information about a system task/function callback		

The VPI routine **vpi_get_cb_info()** shall return information about a simulation-related callback in an **s_cb_data** structure. The memory for this structure shall be allocated by the user.

The **s_cb_data** structure used by **vpi_get_cb_info()** is defined in **vpi_user.h** and is listed in Figure 173.

```
typedef struct t_cb_data

    PLI_INT32      reason;          /* callback reason */
    PLI_INT32      (*cb_rtn)(struct t_cb_data *); /* call routine */
    vpiHandle      obj;            /* trigger object */
    p_vpi_time     time;           /* callback time */
    p_vpi_value     value;         /* trigger object value */
    PLI_INT32      index;          /* index of the memory word or var select
                                   that changed */

    PLI_BYTE8      *user_data;
    s_cb_data, *p_cb_data;
```

Figure 173—The **s_cb_data** structure definition

27.8 vpi_get_data()

vpi_get_data()			
Synopsis:	Get data from an implementation's save/restart location		
Syntax:	vpi_get_data(id, dataLoc, numOfBytes)		
Returns:	Type	Description	
	PLI_INT32	The number of bytes retrieved	
Arguments:	Type	Name	Description
	PLI_INT32	id	A save/restart ID returned from vpi_get(vpiSaveRestartID, NULL)
	PLI_BYTE8 *	dataLoc	Address of application allocated storage
	PLI_INT32	numOfBytes	Number of bytes to be retrieved from save/restart location
Related routines:	Use vpi_put_data() to write saved data		

The routine shall place *numOfBytes* of data into the memory location pointed to by *dataLoc* from a simulation's save/restart location. This memory location has to be properly allocated by the application. The first call for a given *id* will retrieve the data starting at what was placed into the save/restart location with the first call to **vpi_put_data()** for a given *id*. The return value shall be the number of bytes retrieved. On a failure the return value shall be 0. Each subsequent call shall start retrieving data where the last call left off. It shall be a warning for an application to retrieve more data than what was placed into the simulation save/restart location for a given *id*. In this case the *dataLoc* shall be filled with the data that is left for the given *id* and the remaining bytes shall be filled with '\0'. The return value shall be the actual number of bytes retrieved. It shall be acceptable for an application to retrieve less data than what was stored for a given *id* with **vpi_put_data()**. This routine can only be called from a user application routine that has been called for reason **cbStartOfRestart** or **cbEndOfRestart**. The recommended way to get the *id* for **vpi_get_data()** is to

pass it as the value for *user_data* when registering for **cbStartOfRestart** or **cbEndOfRestart** from the **cbStartOfSave** or **cbEndOfSave** user application routine. An application can get the path to the simulations save/restart location by calling **vpi_get_str(vpiSaveRestartLocation, NULL)** from a user application routine that has been called for reason **cbStartOfRestart** or **cbEndOfRestart**.

The following example illustrates using **vpi_get_data()**:

```
/* Uses the global pointer firstWrk */
PLI_INT32 consumer_restart(p_cb_data data)
{
    struct myStruct *wrk; /* myStruct is defined in vpi_put_data()
example */
    PLI_INT32 status;
    PLI_INT32 cnt, size;
    PLI_INT32 id = (PLI_INT32) data->user_data;

    /* Get the number of structures. */
    status = vpi_get_data(id, (PLI_BYTE8 *)&cnt, sizeof(PLI_INT32));
    assert(status > 0); /* Check returned status. */
    size = cnt * sizeof(struct myStruct);

    /* allocate memory for the structures */
    cnt *= sizeof(struct myStruct);
    firstWrk = malloc(cnt);

    /* retrieve the data structures */
    if (cnt != vpi_get_data(id, (PLI_BYTE8 *)firstWrk, cnt))
        return(1); /* error. */

    firstWrk = wrk;
    /* Fix the next pointers in the link list. */
    for (wrk = firstWrk; cnt > 0; cnt--)
    {
        wrk->next = wrk + 1;
        wrk = wrk->next;
    }
    wrk->next = NULL;
    return(SUCCESS);
}
```

27.9 vpi_get_delays()

vpi_get_delays()			
Synopsis:	Retrieve the delays or pulse limits of an object.		
Syntax:	vpi_get_delays(obj, delay_p)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use vpi_put_delays() to set the delays or timing limits of an object		

The VPI routine **vpi_get_delays()** shall retrieve the delays or pulse limits of an object and place them in an `s_vpi_delay` structure that has been allocated by the user. The format of the delay information shall be controlled by the *time_type* flag in the `s_vpi_delay` structure. This routine shall ignore the value of the *type* flag in the `s_vpi_time` structure.

The `s_vpi_delay` and `s_vpi_time` structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in `vpi_user.h` and are listed in Figure 174 and Figure 175.

```

typedef struct t_vpi_delay
{
    struct t_vpi_time *da;          /* pointer to user allocated array of
                                   delay values */
    PLI_INT32 no_of_delays;        /* number of delays */
    PLI_INT32 time_type;          /* [ vpiScaledRealTime, vpiSimTime,
                                   or vpiSuppressTime] */
    PLI_INT32 mtm_flag;           /* true for mtm values */
    PLI_INT32 append_flag;        /* true for append */
    PLI_INT32 pulsere_flag;       /* true for pulsere values */
    s_vpi_delay, *p_vpi_delay;
}

```

Figure 174—The s_vpi_delay structure definition

```

typedef struct t_vpi_time
{
    PLI_INT32 type;               /* [ vpiScaledRealTime, vpiSimTime, vpiSuppressTime] */
    PLI_UINT32 high, low;         /* for vpiSimTime */
    double real;                  /* for vpiScaledRealTime */
    s_vpi_time, *p_vpi_time;
}

```

Figure 175—The s_vpi_time structure definition

The *da* field of the `s_vpi_delay` structure shall be a user-allocated array of `s_vpi_time` structures.

This array shall store delay values returned by **vpi_get_delays()**. The number of elements in this array shall be determined by

- The number of delays to be retrieved
- The **mtm_flag** setting
- The **pulsere_flag** setting

The number of delays to be retrieved shall be set in the *no_of_delays* field of the *s_vpi_delay* structure. Legal values for the number of delays shall be determined by the type of object.

- For primitive objects, the *no_of_delays* value shall be 2 or 3.
- For path delay objects, the *no_of_delays* value shall be 1, 2, 3, 6, or 12.
- For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
- For intermodule path objects, the *no_of_delays* value shall be 2 or 3.

The user allocated *s_vpi_delay* array shall contain delays in the same order in which they occur in the Verilog HDL description. The number of elements for each delay shall be determined by the flags **mtm_flag** and **pulsere_flag**, as shown in Table 213.

Table 213—Size of the *s_vpi_delay->da* array

Flag values	Number of <i>s_vpi_time</i> array elements required for <i>s_vpi_delay->da</i>	Order in which delay elements shall be filled
mtm_flag = FALSE pulsere_flag = FALSE	<i>no_of_delays</i>	1st delay: da[0] -> 1st delay 2nd delay: da[1] -> 2nd delay ...
mtm_flag = TRUE pulsere_flag = FALSE	3 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay 2nd delay: ...
mtm_flag = FALSE pulsere_flag = TRUE	3 * <i>no_of_delays</i>	1st delay: da[0] -> delay da[1] -> reject limit da[2] -> error limit 2nd delay element: ...
mtm_flag = TRUE pulsere_flag = TRUE	9 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay da[3] -> min reject da[4] -> typ reject da[5] -> max reject da[6] -> min error da[7] -> typ error da[8] -> max error 2nd delay: ...

The delay structure has to be allocated before passing a pointer to **vpi_get_delays()**. In the following example, a static structure, **prim_da**, is allocated for use by each call to the **vpi_get_delays()** function.

```
display_prim_delays(prim)
vpiHandle prim;

{
    static s_vpi_time prim_da[ 3];
    static s_vpi_delay delay_s = { NULL, 3, vpiScaledRealTime};
    static p_vpi_delay delay_p = &delay_s;

    delay_s.da = prim_da;
    vpi_get_delays(prim, delay_p);
    vpi_printf("Delays for primitive %s: %6.2f %6.2f %6.2f\n",
        vpi_get_str(vpiFullName, prim)
        delay_p->da[ 0].real, delay_p->da[ 1].real, delay_p->
        >da[ 2].real);
}
```

27.10 vpi_get_str()

vpi_get_str()			
Synopsis:	Get the value of a string property of an object.		
Syntax:	vpi_get_str(prop, obj)		
	Type	Description	
Returns:	PLI_BYTE8 *	Pointer to a character string containing the property value	
	Type	Name	Description
Arguments:	PLI_INT32	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get() to get integer and boolean properties		

The VPI routine **vpi_get_str()** shall return string property values. The string shall be placed in a temporary buffer that shall be used by every call to this routine. If the string is to be used after a subsequent call, the string should be copied to another location. Note that a different string buffer shall be used for string values returned through the **s_vpi_value** structure.

The following example illustrates the usage of **vpi_get_str()**.

```
PLI_BYTE8 *str;
vpiHandle mod = vpi_handle_by_name("top.mod1", NULL);
vpi_printf("Module top.mod1 is an instance of %s\n",
    vpi_get_str(vpiDefName, mod));
```

27.11 vpi_get_systf_info()

vpi_get_systf_info()		
Synopsis:	Retrieve information about a user-defined system task/function-related callback.	
Syntax:	<code>vpi_get_systf_info(obj, systf_data_p)</code>	
Returns:	Type	Description
	void	
Arguments:	Type	Name Description
	vpiHandle	obj Handle to a system task/function-related callback
	p_vpi_systf_data	systf_data_p Pointer to a structure containing callback information
Related routines:	Use <code>vpi_get_cb_info()</code> to retrieve information about a simulation-related callback	

The VPI routine **vpi_get_systf_info()** shall return information about a user-defined system task or function callback in an `s_vpi_systf_data` structure. The memory for this structure shall be allocated by the user.

The `s_vpi_systf_data` structure used by **vpi_get_systf_info()** is defined in `vpi_user.h` and is listed in Figure 176.

```

typedef struct t_vpi_systf_data
{
    PLI_INT32 type;          /* vpiSysTask, vpiSysFunc */
    PLI_INT32 sysfunc_type; /* vpiSysTask, vpi[ Int,Real,Time,Sized,
                                SizedSigned] Func */
    PLI_BYTE8 *tfname;      /* first character must be '$' */
    PLI_INT32 (*calltf) (PLI_BYTE8 *);
    PLI_INT32 (*compiletf) (PLI_BYTE8 *);
    PLI_INT32 (*sizetf) (PLI_BYTE8 *); /* for sized function
                                        callbacks only */
    PLI_BYTE8 *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;

```

Figure 176—The `s_vpi_systf_data` structure definition

27.12 vpi_get_time()

vpi_get_time()			
Synopsis:	Retrieve the current simulation time.		
Syntax:	<code>vpi_get_time(obj, time_p)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_time	time_p	Pointer to a structure containing time information
Related routines:			

The VPI routine **vpi_get_time()** shall retrieve the current simulation time, using the time scale of the object. If *obj* is NULL, the simulation time is retrieved using the simulation time unit. The *time_p->type* field shall be set to indicate if scaled real or simulation time is desired. The memory for the *time_p* structure shall be allocated by the user.

The `s_vpi_time` structure used by **vpi_get_time()** is defined in `vpi_user.h` and is listed in Figure 177 [this is the same time structure as used by **vpi_put_value()**].

```

typedef struct t_vpi_time
{
    PLI_INT32  type;      /* [ vpiScaledRealTime, vpiSimTime, vpiSuppresTime]
    PLI_UINT32 high, low; /* for vpiSimTime */
    double     real;      /* for vpiScaledRealTime */
    s_vpi_time, *p_vpi_time;

```

Figure 177—The s_vpi_time structure definition

27.13 vpi_get_userdata()

vpi_get_userdata()			
Synopsis:	Get user-data value from an implementation's system task/function instance storage location		
Syntax:	<code>vpi_get_userdata(obj)</code>		
Returns:	Type	Description	
	void *	user-data value associated with a system task instance or system function instance	
Arguments:	Type	Name	Description
	vpiHandle	obj	handle to a system task instance or system function instance
Related routines:	Use <code>vpi_put_userdata()</code> to write data into the user data storage area		

This routine shall return the value of the user-data associated with a previous call to **vpi_put_userdata()** for a user-defined system task or function call handle. If no user-data had been previously associated with the object, or if the routine fails, the return value shall be **NULL**.

27.14 vpi_get_value()

vpi_get_value()			
Synopsis:	Retrieve the simulation value of an object.		
Syntax:	vpi_get_value(obj, value_p)		
	Type	Description	
Returns:	void		
	Type	Name	Description
Arguments:	vpiHandle	obj	Handle to an expression
	p_vpi_value	value_p	Pointer to a structure containing value information
Related routines:	Use vpi_put_value() to set the value of an object		

The VPI routine **vpi_get_value()** shall retrieve the simulation value of VPI objects. The value shall be placed in an `s_vpi_value` structure, which has been allocated by the user. The format of the value shall be set by the *format* field of the structure.

When the *format* field is **vpiObjTypeVal**, the routine shall fill in the value and change the *format* field based on the object type, as follows:

- For an integer, **vpiIntVal**
- For a real, **vpiRealVal**
- For a scalar, either **vpiScalar** or **vpiStrength**
- For a time variable, **vpiTimeVal** with **vpiSimTime**
- For a vector, **vpiVectorVal**

The buffer this routine uses for string values shall be different from the buffer that **vpi_get_str()** shall use. The string buffer used by `vpi_get_value()` is overwritten with each call. If the value is needed, it should be saved by the application.

The `s_vpi_value`, `s_vpi_vecval` and `s_vpi_strengthval` structures used by **vpi_get_value()** are defined in `vpi_user.h` and are listed in Figure 178, Figure 179, and Figure 180.

```

ypedef struct t_vpi_value
{
    PLI_INT32 format; /* vpi[ Bin,Oct,Dec,Hex] Str,Scalar,Int,Real,String,
                      Vector,Strength,Suppress,Time,ObjType] Val */
    union
    {
        PLI_BYTE8 *str; /* string value */
        PLI_INT32 scalar; /* vpi[ 0,1,X,Z] */
        PLI_INT32 integer; /* integer value */
        double real; /* real value */
        struct t_vpi_time *time; /* time value */
        struct t_vpi_vecval *vector; /* vector value */
        struct t_vpi_strengthval *strength; /* strength value */
        PLI_BYTE8 *misc; /* ...other */
    } value;
    s_vpi_value, *p_vpi_value;
}

```

Figure 178—The s_vpi_value structure definition

```

ypedef struct t_vpi_vecval
{
    /* following fields are repeated enough times to contain vector */
    PLI_INT32 aval, bval; /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
    s_vpi_vecval, *p_vpi_vecval;
}

```

Figure 179—The s_vpi_vecval structure definition

```

ypedef struct t_vpi_strengthval
{
    PLI_INT32 logic; /* vpi[ 0,1,X,Z] */
    PLI_INT32 s0, s1; /* refer to strength coding in the LRM */
    s_vpi_strengthval, *p_vpi_strengthval;
}

```

Figure 180—The s_vpi_strengthval structure definition

For vectors, the *p_vpi_vecval* field shall point to an array of *s_vpi_vecval* structures. The size of this array shall be determined by the size of the vector, where $array_size = ((vector_size-1)/32 + 1)$. The lsb of the vector shall be represented by the lsb of the 0-indexed element of *s_vpi_vecval* array. The 33rd bit of the vector shall be represented by the lsb of the 1-indexed element of the array, and so on. The memory for the union members *str*, *time*, *vector*, *strength*, and *misc* of the value union in the *s_vpi_value* structure shall be provided by the routine **vpi_get_value()**. This memory shall only be valid until the next call to **vpi_get_value()**. [Note that the user must provide the memory for these members when calling **vpi_put_value()**]. When a value change callback occurs for a value type of **vpiVectorVal**, the system shall create the associated memory (an array of *s_vpi_vecval* structures) and free the memory upon the return of the callback.

Table 214—Return value field of the s_vpi_value structure union

Format	Union member	Return description
vpiBinStrVal	str	String of binary character(s) [1 , 0 , x , z]
vpiOctStrVal	str	String of octal character(s) [0—7 , x , X , z , Z x when all the bits are x X when some of the bits are x z when all the bits are z Z when some of the bits are z
vpiDecStrVal	str	String of decimal character(s) [0—9]
vpiHexStrVal	str	String of hex character(s) [0—f , x , X , z , Z x when all the bits are x X when some of the bits are x z when all the bits are z Z when some of the bits are z
vpiScalarVal	scalar	vpi1 , vpi0 , vpiX , vpiZ , vpiH , vpiL
vpiIntVal	integer	Integer value of the handle. Any bits x or z in the value of the object are mapped to a 0
vpiRealVal	real	Value of the handle as a double
vpiStringVal	str	A string where each 8-bit group of the value of the object is assumed to represent an ASCII character
vpiTimeVal	time	Integer value of the handle using two integers
vpiVectorVal	vector	<i>aval/bval</i> representation of the value of the object
vpiStrengthVal	strength	Value plus strength information
vpiObjTypeVal		Return a value in the closest format of the object

If the format field in the `s_vpi_value` structure is set to **vpiStrengthVal**, the *value.strength* pointer must point to an array of `s_vpi_strengthval` structures. This array must have at least as many elements as there are bits in the vector. If the object is a reg or variable, the strength will always be returned as strong.

If the logic value retrieved by `vpi_get_value()` needs to be preserved for later use, the user must allocate storage and copy the value. The following example can be used to copy a value which was retrieved into an `s_vpi_value` structure into another structure with user-allocated storage.

```

/*
 * Copy s_vpi_value structure - must first allocate pointed to
 * fields.
 * nvalp must be previously allocated.
 * Need to first determine size for vector value.
 */
void copy_vpi_value(s_vpi_value *nvalp, s_vpi_value *ovalp,
                   PLI_INT32 blen, PLI_INT32 nd_alloc)
{
    int i;
    PLI_INT32 numvals;
    nvalp->format = ovalp->format;
    switch (nvalp->format) {
        /* all string values */
        case vpiBinStrVal: case vpiOctStrVal: case vpiDecStrVal:

```

```

        case vpiHexStrVal: case vpiStringVal:
            if (nd_alloc) nvalp->value.str = malloc(strlen(ovalp->value.str)+1);
            strcpy(nvalp->value.str, ovalp->value.str);
            break;
        case vpiScalarVal:
            nvalp->value.scalar = ovalp->value.scalar;
            break;
        case vpiIntVal:
            nvalp->value.integer = ovalp->value.integer;
            break;
        case vpiRealVal:
            nvalp->value.real = ovalp->value.real;
            break;
        case vpiVectorVal:
            numvals = (blen + 31) >> 5;
            if (nd_alloc)
            {
                nvalp->value.vector = (p_vpi_vecval)
                    malloc(numvals*sizeof(s_vpi_vecval));
            }
            /* t_vpi_vecval is really array of the 2 integer a/b sections */
            /* memcpy or bcopy better here */
            for (i = 0; i < numvals; i++)
                nvalp->value.vector[i] = ovalp->value.vector[i];
            break;
        case vpiStrengthVal:
            if (nd_alloc)
            {
                nvalp->value.strength = (p_vpi_strengthval)
                    malloc(sizeof(s_vpi_strengthval));
            }
            /* assumes C compiler supports struct assign */
            *(nvalp->value.strength) = *(ovalp->value.strength);
            break;
        case vpiTimeVal:
            nvalp->value.time = (p_vpi_time) malloc(sizeof(s_vpi_time));
            /* assumes C compiler supports struct assign */
            *(nvalp->value.time) = *(ovalp->value.time);
            break;
        /* not sure what to do here? */
        case vpiObjTypeVal: case vpiSuppressVal:
            vpi_printf(
                "***ERR: can not copy vpiObjTypeVal or vpiSuppressVal formats -
not for filled records.\n");
            break;
    }
}

```

To get the ASCII values of UDP table entries (see Table 40), the *p_vpi_vecval* field shall point to an array of *s_vpi_vecval* structures. The size of this array shall be determined by the size of the table entry (no. of symbols per table entry), where *array_size* = $((\text{table_entry_size}-1)/4 + 1)$. Each symbol shall require two bytes; the ordering of the symbols within *s_vpi_vecval* shall be the most significant byte of *abit* first, then the least significant byte of *abit*, then the most significant byte of *bbit* and then the least significant byte of *bbit*. Each symbol can be either one or two characters; when it is a single character, the second byte of the pair shall be an ASCII \0.

Real valued objects shall be converted to an integer using the rounding defined in 3.9.2 before being returned in a format other than **vpiRealVal** and **vpiStringVal**. If the format specified is **vpiStringVal** then the value shall be returned as a string representation of a floating point number. The format of this string shall be in decimal notation with at most 16 digits of precision.

If a constant object's **vpiConstType** is **vpiStringVal**, the value shall be retrieved using either a format of **vpiStringVal** or **vpiVectorVal**.

The *misc* field in the `s_vpi_value` structure shall provide for alternative value types, which can be implementation specific. If this field is utilized, one or more corresponding format types shall also be provided.

In the following example, the binary value of each net that is contained in a particular module and whose name begins with a particular string is displayed. [This function makes use of the `strcmp()` facility normally declared in a `string.h` C library.]

```
void display_certain_net_values(mod, target)
vpiHandle mod;
PLI_BYTE8 *target;
{
    static s_vpi_value value_s = { vpiBinStrVal };
    static p_vpi_value value_p = &value_s;
    vpiHandle net, itr;

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        PLI_BYTE8 *net_name = vpi_get_str(vpiName, net);
        if (strcmp(target, net_name) == 0)
        {
            vpi_get_value(net, value_p);
            vpi_printf("Value of net %s: %s\n",
                vpi_get_str(vpiFullName, net), value_p->value.str);
        }
    }
}
```

The following example illustrates the use of `vpi_get_value()` to access UDP table entries. Two sample outputs from this example are provided after the example.

```
/*
 * hUDP must be a handle to a UDP definition
 */
static void dumpUDPTTableEntries(vpiHandle hUDP)
{
    vpiHandle hEntry, hEntryIter;
    s_vpi_value value;
    PLI_INT32 numb;
    PLI_INT32 udpType;
    PLI_INT32 item;
    PLI_INT32 entryVal;
    PLI_INT32 *abItem;
    PLI_INT32 cnt, cnt2;
    numb = vpi_get(vpiSize, hUDP);
    udpType = vpi_get(vpiPrimType, hUDP);
    if (udpType == vpiSeqPrim)
        numb++; /* There is one more table entry for state */
    numb++; /* There is a table entry for the output */
    hEntryIter = vpi_iterate(vpiTableEntry, hUDP);
```

```

    if (!hEntryIter)
        return;
    value.format = vpiVectorVal;
    while(hEntry = vpi_scan(hEntryIter))
    {
        vpi_printf("\n");
        /* Show the entry as a string */
        value.format = vpiStringVal;
        vpi_get_value(hEntry, &value);
        vpi_printf("%s\n", value.value.str);
        /* Decode the vector value format */
        value.format = vpiVectorVal;
        vpi_get_value(hEntry, &value);
        abItem = (PLI_INT32 *)value.value.vector;
        for(cnt=((numb-1)/2+1);cnt>0;cnt--)
        {
            entryVal = *abItem;
            abItem++;
            /* Rip out 4 characters */
            for (cnt2=0;cnt2<4;cnt2++)
            {
                item = entryVal&0xff;
                if (item)
                    vpi_printf("%c", item);
                else
                    vpi_printf("_");
                entryVal = entryVal>>8;
            }
        }
        vpi_printf("\n");
    }
}

```

For a UDP table of:

```

1      0      :?:1;
0      (01)   :?:-;
(10)  0       :0:1;

```

The output from the preceding example would be:

```

10:1
_0_1___1
01:0
_1_0___0
00:1
_0_0___1

```

For a UDP table entry of:

```

1      0      :?:1;
0      (01)   :?:-;
(10)  0       :0:1;

```

The output from the preceding example would be:

```
10:?:1
 0 1 1 ?
0 (01):?:-
10 0 - ?
(10) 0:0:1
_001_1_0
```

27.15 vpi_get_vlog_info()

vpi_get_vlog_info()			
Synopsis:	Retrieve information about Verilog simulation execution.		
Syntax:	vpi_get_vlog_info(vlog_info_p)		
Type		Description	
Returns:	PLI_INT32	1 (true) on success and 0 (false) on failure	
Type		Name	Description
Arguments:	p_vpi_vlog_info	vlog_info_p	Pointer to a structure containing simulation information

The VPI routine **vpi_get_vlog_info()** shall obtain the following information about Verilog product execution:

- The number of invocation options (*argc*)
- Invocation option values (*argv*)
- Product and version strings

The information shall be contained in an `s_vpi_vlog_info` structure. The routine shall return 1 (true) on success and 0 (false) on failure.

The `s_vpi_vlog_info` structure used by **vpi_get_vlog_info()** is defined in `vpi_user.h` and is listed in Figure 181.

```
typedef struct t_vpi_vlog_info
{
    PLI_INT32 argc;
    PLI_BYTE8 **argv;
    PLI_BYTE8 *product;
    PLI_BYTE8 *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;
```

Figure 181—The s_vpi_vlog_info structure definition

The format of the *argv* array is that each pointer in the array shall point to a `NULL` terminated character array which contains the string located on the tool's invocation command line. There shall be *argc* entries in the *argv* array. The value in entry zero shall be the tool's name.

The argument following a `-f` argument shall contain a pointer to a `NULL` terminated array of pointers to characters. This new array shall contain the parsed contents of the file. The value in entry zero shall contain the name of the file. The remaining entries shall contain pointers to `NULL` terminated character arrays containing the different options in the file. The last entry in this array shall be a `NULL`. If one of the options is a `-f` then the next pointer shall behave the same as described above.

27.16 vpi_handle()

vpi_handle()			
Synopsis:	Obtain a handle to an object with a one-to-one relationship.		
Syntax:	<code>vpi_handle(type, ref)</code>		
Type		Description	
Returns:	<code>vpiHandle</code>	Handle to an object	
Type		Name	Description
Arguments:	<code>PLI_INT32</code>	<code>type</code>	An integer constant representing the type of object for which to obtain a handle
	<code>vpiHandle</code>	<code>ref</code>	Handle to a reference object
Related routines:	Use <code>vpi_iterate()</code> and <code>vpi_scan()</code> to obtain handles to objects with a one-to-many relationship Use <code>vpi_handle_multi()</code> to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_handle()** shall return the object of type *type* associated with object *ref*. The one-to-one relationships that are traversed with this routine are indicated as single arrows in the data model diagrams.

The following example application displays each primitive that an input net drives.

```
void display_driven_primitives(net)
vpiHandle net;
{
    vpiHandle load, prim, itr;
    vpi_printf("Net %s drives terminals of the primitives: \n",
        vpi_get_str(vpiFullName, net));
    itr = vpi_iterate(vpiLoad, net);
    if (!itr)
        return;
    while (load = vpi_scan(itr))
    {
        switch(vpi_get(vpiType, load))
        {
            case vpiGate:
            case vpiSwitch:
            case vpiUdp:
                prim = vpi_handle(vpiPrimitive, load);
                vpi_printf("\t%s\n", vpi_get_str(vpiFullName,
prim));
            }
        }
    }
}
```

27.17 vpi_handle_by_index()

vpi_handle_by_index()			
Synopsis:	Get a handle to an object using its index number within a parent object.		
Syntax:	<code>vpi_handle_by_index(obj, index)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	PLI_INT32	index	Index number of the object for which to obtain a handle

The VPI routine **vpi_handle_by_index()** shall return a handle to an object based on the index number of the object within a parent object. This function can be used to access all objects that can access an expression using **vpiIndex**. Argument *obj* shall represent the parent of the indexed object. For example, to access a net-bit, *obj* would be the associated net, while for a memory word, *obj* would be the associated memory.

27.18 vpi_handle_by_multi_index()

vpi_handle_by_multi_index()			
Synopsis:	Obtain a handle to a sub object using an array of indexes and a parent object.		
Syntax:	<code>vpi_handle_by_multi_index(obj, num_index, index_array)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object of type vpiRegBit, vpiNetBit, vpiRegWord, or vpiNetWord	
Arguments:	Type	Name	Description
	vpiHandle	obj	handle to an object
	PLI_INT32	num_index	number of indexes in the index array
	PLI_INT32 *	index_array	array of indexes. Left most index first
Related routines:			

The VPI routine **vpi_handle_by_multi_index()** shall return a handle to an object based on the list of indexes and parent object passed in. The argument *num_index* will contain the number of indexes in the provided array *index_array*. The order of the indexes provided, shall be for the left most select first, progressing to the right most select last. This function can be used to access all objects whose property **vpiMultiArray** is TRUE. This routine shall only provide access to a bit or word of the parent object.

27.19 vpi_handle_by_name()

vpi_handle_by_name()			
Synopsis:	Get a handle to an object with a specific name.		
Syntax:	<code>vpi_handle_by_name(name, scope)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	PLI_BYTE8 *	name	A character string or pointer to a string containing the name of an object
	vpiHandle	scope	Handle to a Verilog HDL scope

The VPI routine **vpi_handle_by_name()** shall return a handle to an object with a specific name. This function can be applied to all objects with a *fullname* property. The *name* can be hierarchical or simple. If *scope* is `NULL`, then *name* shall be searched for from the top level of hierarchy. If a scope object is provided, then search within that scope only.

27.20 vpi_handle_multi()

vpi_handle_multi()			
Synopsis:	Obtain a handle for an object in a many-to-one relationship.		
Syntax:	<code>vpi_handle_multi(type, ref1, ref2, ...)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	PLI_INT32	type	An integer constant representing the type of object for which to obtain a handle
	vpiHandle	ref1, ref2, ...	Handles to two or more reference objects
Related routines:	Use <code>vpi_iterate()</code> and <code>vpi_scan()</code> to obtain handles to objects with a one-to-many relationship Use <code>vpi_handle()</code> to obtain handles to objects with a one-to-one relationship		

The VPI routine **vpi_handle_multi()** can be used to return a handle to an object of type **vpiInterModPath** associated with a list of *output port* and *input port* reference objects. The ports shall be of the same size and can be at different levels of the hierarchy.

27.21 vpi_iterate()

vpi_iterate()			
Synopsis:	Obtain an iterator handle to objects with a one-to-many relationship.		
Syntax:	<code>vpi_iterate(type, ref)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an iterator for an object	
Arguments:	Type	Name	Description
	PLI_INT32	type	An integer constant representing the type of object for which to obtain iterator handles
	vpiHandle	ref	Handle to a reference object
Related routines:	Use <code>vpi_scan()</code> to traverse the HDL hierarchy using the iterator handle returned from <code>vpi_iterate()</code> Use <code>vpi_handle()</code> to obtain handles to object with a one-to-one relationship Use <code>vpi_handle_multi()</code> to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_iterate()** shall be used to traverse one-to-many relationships, which are indicated as double arrows in the data model diagrams. The **vpi_iterate()** routine shall return a handle to an iterator, whose type shall be **vpiIterator**, which can be used by **vpi_scan()** to traverse all objects of type *type* associated with object *ref*. To get the reference object from the iterator object use **vpi_handle(vpiUse, iterator_handle)**. If there are no objects of type *type* associated with the reference handle *ref*, then the **vpi_iterate()** routine shall return `NULL`.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```
void display_nets(mod)
vpiHandle mod;
{
    vpiHandle net;
    vpiHandle itr;

    vpi_printf("Nets declared in module %s\n",
        vpi_get_str(vpiFullName, mod));

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        vpi_printf("\t%s", vpi_get_str(vpiName, net));
        if (vpi_get(vpiVector, net))
        {
            vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
        }
        else vpi_printf("\n");
    }
}
```

27.22 vpi_mcd_close()

vpi_mcd_close()			
Synopsis:	Close one or more files opened by vpi_mcd_open().		
Syntax:	vpi_mcd_close(<i>mcd</i>)		
	Type	Description	
Returns:	PLI_UINT32	0 if successful, the <i>mcd</i> of unclosed channels if unsuccessful	
	Type	Name	Description
Arguments:	PLI_UINT32	<i>mcd</i>	A multi-channel descriptor representing the files to close
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file Use vpi_mcd_flush() to flush a file output buffer Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_close()** shall close the file(s) specified by a multi-channel descriptor, *mcd*. Several channels can be closed simultaneously, since channels are represented by discrete bits in the integer *mcd*. On success this routine shall return a 0; on error it shall return the *mcd* value of the unclosed channels. This routine can also be used to close file descriptors which were opened using the system function `$fopen()`. See 17.2.1 for the functional description of `$fopen()`.

The following descriptors are predefined, and cannot be closed using vpi_mcd_close():

descriptor 1 is for the output channel of the software product which invoked the PLI application and the current log file

27.23 vpi_mcd_flush()

vpi_mcd_flush()			
Synopsis:	Flushes the data from the given MCD output buffers.		
Syntax:	vpi_mcd_flush(<i>mcd</i>)		
	Type	Description	
Returns:	PLI_INT32	0 if successful, non-zero if unsuccessful	
	Type	Name	Description
Arguments:	PLI_UINT32	<i>mcd</i>	A multi-channel descriptor representing the files to which to write
Related routines:	Use vpi_mcd_printf() to write a finite number of arguments to an opened file Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close a file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The routine **vpi_mcd_flush()** shall flush the output buffers for the file(s) specified by the multi-channel descriptor, *mcd*.

27.24 vpi_mcd_name()

vpi_mcd_name()			
Synopsis:	Get the name of a file represented by a channel descriptor.		
Syntax:	<code>vpi_mcd_name (cd)</code>		
	Type	Description	
Returns:	PLI_BYTE8 *	Pointer to a character string containing the name of a file	
	Type	Name	Description
Arguments:	PLI_UINT32	cd	A channel descriptor representing a file
Related routines:	Use <code>vpi_mcd_open()</code> to open a file Use <code>vpi_mcd_close()</code> to close files Use <code>vpi_mcd_printf()</code> to write to an opened file Use <code>vpi_mcd_flush()</code> to flush a file output buffer Use <code>vpi_mcd_vprintf()</code> to write a variable number of arguments to an opened file		

The VPI routine **vpi_mcd_name()** shall return the name of a file represented by a single-channel descriptor, *cd*. On error, the routine shall return `NULL`. This routine shall overwrite the returned value on subsequent calls. If the application needs to retain the string, it should copy it. This routine can be used to get the name of any file, opened using the system function `$fopen` or the VPI routine **vpi_mcd_open()**. The channel descriptor *cd* could be an `fd` file descriptor returned from `$fopen` (indicated by the most significant bit being set) or an *mcd* multi-channel descriptor returned by either the system function `$fopen` or the VPI routine **vpi_mcd_open()**. See 17.2.1 for the functional description of `$fopen`.

27.25 vpi_mcd_open()

vpi_mcd_open()			
Synopsis:	Open a file for writing.		
Syntax:	<code>vpi_mcd_open (file)</code>		
	Type	Description	
Returns:	PLI_UINT32	A multi-channel descriptor representing the file that was opened	
	Type	Name	Description
Arguments:	PLI_BYTE8 *	file	A character string or pointer to a string containing the file name to be opened
Related routines:	Use <code>vpi_mcd_close()</code> to close a file Use <code>vpi_mcd_printf()</code> to write to an opened file Use <code>vpi_mcd_vprintf()</code> to write a variable number of arguments to an opened file Use <code>vpi_mcd_flush()</code> to flush a file output buffer Use <code>vpi_mcd_name()</code> to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_open()** shall open a file for writing and shall return a corresponding multi-channel description number (mcd). The channel descriptor 1 (least significant bit) is reserved for representing the output channel of the software product which invoked the PLI application and the log file (if one is currently open). The channel descriptor 32 (most significant bit) is reserved to represent a file descriptor (fd) returned from the Verilog HDL \$fopen system function.

The mcd descriptor returned by **vpi_mcd_open()** routine is compatible with the mcd descriptors returned from the \$fopen system function. The mcd descriptors returned from **vpi_mcd_open()** and from \$fopen may be shared between the HDL system tasks which use mcd descriptors and the VPI routines which use mcd descriptors. Note that the \$fopen system function can also return fd file descriptors (indicated by the most significant bit being set). An fd is not compatible with the mcd descriptor returned by **vpi_mcd_open()**. See 17.2.1 for the functional description of \$fopen.

The **vpi_mcd_open()** routine shall return a 0 on error. If the file has already been opened either by a previous call to **vpi_mcd_open()** or using \$fopen in the Verilog source code, then **vpi_mcd_open()**, shall return the descriptor number.

27.26 vpi_mcd_printf()

vpi_mcd_printf()			
Synopsis:	Write to one or more files opened with vpi_mcd_open() or \$fopen.		
Syntax:	vpi_mcd_printf(mcd, format, ...)		
		Type	Description
Returns:	PLI_INT32	The number of characters written	
Arguments:	Type	Name	Description
	PLI_UINT32	mcd	A multi-channel descriptor representing the files to which to write
	PLI_BYTE8 *	format	A format string using the C fprintf() format
Related routines:	Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close a file Use vpi_mcd_flush() to flush a file output buffer Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_printf()** shall write to one or more channels (up to 31) determined by the mcd. An mcd of 1 (bit 0 set) corresponds to the channel 1, an mcd of 2 (bit 1 set) corresponds to channel 2, an mcd of 4 (bit 2 set) corresponds to channel 3, and so on. Channel 1 is reserved for the output channel of the software product which invoked the PLI application and the current log file. The most significant bit of the descriptor is reserved by the tool to indicate that the descriptor is actually a file descriptor instead of an mcd. **vpi_mcd_printf()** shall also write to a file represented by an mcd which was returned from the Verilog HDL \$fopen system function. **vpi_mcd_printf()** shall not write to a file represented by an fd file descriptor returned from \$fopen (indicated by the most significant bit being set). See 17.2.1 for the functional description of \$fopen

Several channels can be written to simultaneously, since channels are represented by discrete bits in the integer mcd.

The text written shall be controlled by one or more format strings. The format strings shall use the same format as the C fprintf() routine. The routine shall return the number of characters printed, or EOF if an error occurred.

27.27 vpi_mcd_vprintf()

vpi_mcd_vprintf()			
Synopsis:	Write to one or more files opened with vpi_mcd_open() or \$fopen using varargs which are already started.		
Syntax:	vpi_mcd_vprintf(mcd, format, ap)		
Returns:	Type	Description	
	PLI_INT32	The number of characters written	
Arguments:	Type	Name	Description
	PLI_UINT32	mcd	A multi-channel descriptor representing the files to which to write
	PLI_BYTE8 *	format	A format string using the C printf() format
	va_list	ap	An already started varargs list
Related routines:	Use vpi_mcd_printf() to write a finite number of arguments to an opened file Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close a file Use vpi_mcd_flush() to flush a file output buffer Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

This routine performs the same function as **vpi_mcd_printf()**, except that varargs has already been started.

27.28 vpi_printf()

vpi_printf()			
Synopsis:	Write to the output channel of the software product which invoked the PLI application and the current product log file.		
Syntax:	vpi_printf(format, ...)		
	Type	Description	
Returns:	PLI_INT32	The number of characters written	
	Type	Name	Description
Arguments:	PLI_BYTE8 *	format	A format string using the C printf() format
Related routines:	Use vpi_vprintf() to write a variable number of arguments Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_flush() to flush a file output buffer Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file		

The VPI routine **vpi_printf()** shall write to both the output channel of the software product which invoked the PLI application and the current product log file. The format string shall use the same format as the C `printf()` routine. The routine shall return the number of characters printed, or EOF if an error occurred.

27.29 vpi_put_data()

vpi_put_data()			
Synopsis:	Put data into an implementation's save/restart location		
Syntax:	vpi_put_data(id, dataLoc, numOfBytes)		
Returns:	Type	Description	
	PLI_INT32	The number of bytes written	
Arguments:	Type	Name	Description
	PLI_INT32	id	A save/restart ID returned from vpi_get(vpiSaveRestartID, NULL)
	PLI_BYTE8 *	dataLoc	Address of application allocated storage
	PLI_INT32	numOfBytes	Number of bytes to be added to save/restart location
Related routines:	Use vpi_get_data() to retrieve saved data		

This routine shall place **numOfBytes**, which must be greater than zero, of data located at **dataLoc** into an implementation's save/restart location. The return value shall be the number of bytes written. A zero shall be returned if an error is detected. There shall be no restrictions:

- on how many times the routine can be called for a given *id*.
- on the order applications put data using the different *ids*.

The data from multiple calls to **vpi_put_data()** with the same *id* shall be stored by the simulator in such a way that the opposing routine **vpi_get_data()** can pull data out of the save/restart location using different size chunks. This routine can only be called from a user application routine that has been called for the reason **cbStartOfSave** or **cbEndOfSave**. A user can get the path to the implementation's save/restart location by calling **vpi_get_str(vpiSaveRestartLocation, NULL)** from a user application routine that has been called for reason **cbStartOfSave** or **cbEndOfSave**.

The following example illustrates using **vpi_put_data()**:

```
/* example of how to place data into a save/restart location */
struct myStruct{
    struct myStruct *next;
    PLI_INT32 d1;
    PLI_INT32 d2;
}
struct myStruct *firstWrk; /* This data structure created
elsewhere. */

PLI_INT32 consumer_save(p_cb_data data)
{
    struct myStruct *wrk;
    s_cb_data cbData;
    vpiHandle cbHdl;
    PLI_INT32 id = 0;
    PLI_INT32 cnt = 0;
```

```

/* Get the number of structures */
wrk = firstWrk;
while (wrk)
{
    cnt++;
    wrk = wrk->next;
}

/* now save the data */
wrk = firstWrk;

/* save the number of data structures */
id = vpi_get(vpiSaveRestartID, NULL);

/*
 * save the different data structures. Please note that
 * a pointer is being saved. While this is allowed an
 * application must change it to something useful on a restart.
 */
while (wrk)
{
    wrk = wrk->next;
}

/* register a call for restart */

/*
 * We need the "id" so that the saved data can be retrieved.
 * Using the user_data field of the callback structure is the
 * easiest way to pass this information to retrieval
operation.
 */
cbData.user_data = (PLI_BYTE8 *)id;
cbData.reason = cbStartOfRestart;
cbData.cb_rtn = consumer_restart; /*
vpi_get_data()                * Please see
this                           * for a description of
                               * routine.
                               */

cbData.value = NULL;
cbData.time = NULL;
cbHdl = vpi_register_cb(&cbData);
vpi_free_object(cbHdl);
return(0);
}

```

27.30 vpi_put_delays()

vpi_put_delays()			
Synopsis:	Set the delays or timing limits of an object.		
Syntax:	vpi_put_delays(obj, delay_p)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use vpi_get_delays() to retrieve delays or timing limits of an object		

The VPI routine **vpi_put_delays()** shall set the delays or timing limits of an object as indicated in the *delay_p* structure. The same ordering of delays shall be used as described in the **vpi_get_delays()** function. If only the delay changes, and not the pulse limits, the pulse limits shall retain the values they had before the delays were altered.

The *s_vpi_delay* and *s_vpi_time* structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in *vpi_user.h* and are listed in Figure 182 and Figure 183.

```
typedef struct t_vpi_delay
{
    struct t_vpi_time *da; /* pointer to user allocated array of delay values */
    'LI_INT32 no_of_delays; /* number of delays */
    'LI_INT32 time_type;    /* [ vpiScaledRealTime, vpiSimTime, vpiSuppressTime] */
    'LI_INT32 mtm_flag;     /* true for mtm values */
    'LI_INT32 append_flag;  /* true for append */
    'LI_INT32 pulserere_flag; /* true for pulserere values */
    s_vpi_delay, *p_vpi_delay;
}
```

Figure 182—The s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
    'LI_INT32 type; /* [ vpiScaledRealTime, vpiSimTime, vpiSuppressTime] */
    'LI_UINT32 high, low; /* for vpiSimTime */
    double real; /* for vpiScaledRealTime */
    s_vpi_time, *p_vpi_time;
}
```

Figure 183—The s_vpi_time structure definition

The *da* field of the *s_vpi_delay* structure shall be a user-allocated array of *s_vpi_time* structures. This array stores the delay values to be written by **vpi_put_delays()**. The number of elements in this array is determined by:

The number of delays to be written
 The **mtm_flag** setting
 The **pulsere_flag** setting

The number of delays to be set shall be set in the *no_of_delays* field of the *s_vpi_delay* structure. Legal values for the number of delays shall be determined by the type of object.

For primitive objects, the *no_of_delays* value shall be 2 or 3.
 For path delay objects, the *no_of_delays* value shall be 1, 2, 3, 6, or 12.
 For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
 For intermodule path objects, the *no_of_delays* value shall be 2 or 3.

The user allocated *s_vpi_delay* array shall contain delays in the same order in which they occur in the Verilog HDL description. The number of elements for each delay shall be determined by the flags **mtm_flag** and **pulsere_flag**, as shown in Table 215.

Table 215—Size of the *s_vpi_delay*->*da* array

Flag values	Number of <i>s_vpi_time</i> array elements required for <i>s_vpi_delay</i> -> <i>da</i>	Order in which delay elements shall be filled
mtm_flag = FALSE pulsere_flag = FALSE	<i>no_of_delays</i>	1st delay: <i>da</i> [0] -> 1st delay 2nd delay: <i>da</i> [1] -> 2nd delay ...
mtm_flag = TRUE pulsere_flag = FALSE	3 * <i>no_of_delays</i>	1st delay: <i>da</i> [0] -> min delay <i>da</i> [1] -> typ delay <i>da</i> [2] -> max delay 2nd delay: ...
mtm_flag = FALSE pulsere_flag = TRUE	3 * <i>no_of_delays</i>	1st delay: <i>da</i> [0] -> delay <i>da</i> [1] -> reject limit <i>da</i> [2] -> error limit 2nd delay element: ...
mtm_flag = TRUE pulsere_flag = TRUE	9 * <i>no_of_delays</i>	1st delay: <i>da</i> [0] -> min delay <i>da</i> [1] -> typ delay <i>da</i> [2] -> max delay <i>da</i> [3] -> min reject <i>da</i> [4] -> typ reject <i>da</i> [5] -> max reject <i>da</i> [6] -> min error <i>da</i> [7] -> typ error <i>da</i> [8] -> max error 2nd delay: ...

The following example application accepts a module path handle, rise and fall delays, and replaces the delays of the indicated path.

```
void set_path_rise_fall_delays(path, rise, fall)
vpiHandle path;
double rise, fall;
{
    static s_vpi_time path_da[ 2] ;
    static s_vpi_delay delay_s = { NULL, 2, vpiScaledRealTime} ;
    static p_vpi_delay delay_p = &delay_s;

    delay_s.da = path_da;
    path_da[ 0].real = rise;
    path_da[ 1].real = fall;
    vpi_put_delays(path, delay_p);
}
```

27.31 vpi_put_userdata()

vpi_put_userdata()			
Synopsis:	Put user-data value into an implementation’s system task/function instance storage location		
Syntax:	vpi_put_userdata (obj, userdata)		
Returns:	Type	Description	
	PLI_INT32	returns 1 on success and 0 if an error occurs	
Arguments:	Type	Name	Description
	vpiHandle	obj	handle to a system task instance or system function instance
	void *	userdata	user-data value to be associated with the system task instance or system function instance
Related routines:	Use vpi_get_userdata() to retrieve the user-data value		

This routine will associate the value of the input *userdata* with the specified user-defined system task or function call handle. The stored value can later be retrieved with the routine **vpi_get_userdata()**. The routine will return a value of 1 on success or a 0 if it fails.

27.32 vpi_put_value()

vpi_put_value()			
Synopsis:	Set a value on an object.		
Syntax:	<code>vpi_put_value(obj, value_p, time_p, flags)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to the scheduled event caused by vpi_put_value()	
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_value	value_p	Pointer to a structure with value information
	p_vpi_time	time_p	Pointer to a structure with delay information
	PLI_INT32	flags	Integer constants that set the delay mode
Related routines:	Use vpi_get_value() to retrieve the value of an expression		

The VPI routine **vpi_put_value()** shall set simulation logic values on an object. The value to be set shall be stored in an `s_vpi_value` structure that has been allocated. The legal values which may be specified for each value format are listed in Table 214. The delay time before the value is set shall be stored in an `s_vpi_time` structure that has been allocated. The routine can be applied to nets, regs, variables, variable selects, memory words, named events, system function calls, sequential UDPs, and scheduled events. The *flags* argument shall be used to direct the routine to use one of the following delay modes:

vpiInertialDelay	All scheduled events on the object shall be removed before this event is scheduled.
vpiTransportDelay	All events on the object scheduled for times later than this event shall be removed (modified transport delay).
vpiPureTransportDelay	No events on the object shall be removed (transport delay).
vpiNoDelay	The object shall be set to the passed value with no delay. Argument <i>time_p</i> shall be ignored and can be set to NULL.
vpiForceFlag	The object shall be forced to the passed value with no delay (same as the Verilog HDL procedural force). Argument <i>time_p</i> shall be ignored and can be set to NULL.
vpiReleaseFlag	The object shall be released from a forced value (same as the Verilog HDL procedural release). Argument <i>time_p</i> shall be ignored and can be set to NULL. The <i>value_p</i> shall be updated with the value of the object after its release.
vpiCancelEvent	A previously scheduled event shall be cancelled. The object passed to vpi_put_value() shall be a handle to an object of type vpiSchedEvent .

If the *flags* argument also has the bit mask **vpiReturnEvent**, **vpi_put_value()** shall return a handle of type **vpiSchedEvent** to the newly scheduled event, provided there is some form of a delay and an event is scheduled. If the bit mask is not used, or if no delay is used, or if an event is not scheduled, the return value shall be NULL.

The handle to the event can be cancelled by calling **vpi_put_value()** with the flag set to **vpiCancelEvent**. The *value_p* and *time_p* arguments to **vpi_put_value()** are not needed for cancelling an event, and can be set to **NULL**. It shall not be an error to cancel an event that has already occurred. The scheduled event can be tested by calling **vpi_get()** with the flag **vpiScheduled**. If an event is cancelled, it shall simply be removed from the event queue. Any effects that were caused by scheduling the event shall remain in effect (e.g., events that were cancelled due to inertial delay).

Calling **vpi_free_object()** on the handle shall free the handle but shall not affect the event.

When **vpi_put_value()** is called for an object of type **vpiNet** or **vpiNetBit**, and with modes of **vpiInertialDelay**, **vpiTransportDelay**, **vpiPureTransportDelay**, or **vpiNoDelay**, the value supplied overrides the resolved value of the net. This value shall remain in effect until one of the drivers of the net changes value. When this occurs, the net shall be re-evaluated using the normal resolution algorithms.

It shall be illegal to specify the format of the value as **vpiStringVal** when putting a value to a real variable or a system function call of type **vpiRealFunc**. It shall be illegal to specify the format of the value as **vpiStrengthVal** when putting a value to a vector object.

When **vpi_put_value()** with a **vpiForce** flag is used, it shall perform a procedural force of a value onto the same types of objects as supported by a procedural force. A **vpiRelease** flag shall release the forced value. This shall be the same functionality as the procedural force and release keywords in the Verilog HDL (refer to 9.3.2).

Sequential UDPs shall be set to the indicated value with no delay regardless of any delay on the primitive instance. Putting values to UDP instances must be done using the **vpiNoDelay** flag. Attempting to use the other delay modes shall result in an error.

Calling **vpi_put_value()** on an object of type **vpiNamedEvent** shall cause the named event to toggle. Objects of type **vpiNamedEvent** shall not require an actual value and the *value_p* argument may be **NULL**.

The **vpi_put_value()** routine shall also return the value of a system function by passing a handle to the user-defined system function as the object handle. This should only occur during execution of the calltf routine for the system function. Attempts to use **vpi_put_value()** with a handle to the system function when the calltf routine is not active shall be ignored. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 will be applied. Putting return values to system functions must be done using the **vpiNoDelay** flag.

The **vpi_put_value()** routine shall only return a system function value in a calltf application, when the call to the system function is active. The action of **vpi_put_value()** to a system function shall be ignored when the system function is not active. Putting values to system function must be done using the **vpiNoDelay** flag.

The *s_vpi_value* and *s_vpi_time* structures used by **vpi_put_value()** are defined in *vpi_user.h* and are listed in Figure 184 and Figure 185.

```

ypedef struct t_vpi_value
{
    PLI_INT32 format; /* vpi[ Bin,Oct,Dec,Hex] Str,Scalar,Int,Real,String,
                        Vector,Strength,Suppress,Time,ObjType] Val */
    union
    {
        PLI_BYTE8 *str; /* string value */
        PLI_INT32 scalar; /* vpi[ 0,1,X,Z] */
        PLI_INT32 integer; /* integer value */
        double real; /* real value */
        struct t_vpi_time *time; /* time value */
        struct t_vpi_vecval *vector; /* vector value */
        struct t_vpi_strengthval *strength; /* strength value */
        PLI_BYTE8 *misc; /* ...other */
    } value;
} s_vpi_value, *p_vpi_value;

```

Figure 184—The s_vpi_value structure definition

```

pedef struct t_vpi_time
{
    PLI_INT32 type; /* [ vpiScaledRealTime, vpiSimTime, vpiSuppressTime] */
    PLI_UINT32 high, low; /* for vpiSimTime */
    double real; /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;

```

Figure 185—The s_vpi_time structure definition

```

pedef struct t_vpi_vecval
{
    /* following fields are repeated enough times to contain vector */
    PLI_INT32 aval, bval; /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;

```

Figure 186—The s_vpi_vecval structure definition

```

ypedef struct t_vpi_strengthval
{
    PLI_INT32 logic; /* vpi[ 0,1,X,Z] */
    PLI_INT32 s0, s1; /* refer to strength coding below */
} s_vpi_strengthval, *p_vpi_strengthval;

```

Figure 187—The s_vpi_strengthval structure definition

For **vpiScaledRealTime**, the indicated time shall be in the timescale associated with the object.

27.33 vpi_register_cb()

vpi_register_cb()			
Synopsis:	Register simulation-related callbacks.		
Syntax:	<code>vpi_register_cb(cb_data_p)</code>		
Returns:	Type	Description	
	<code>vpiHandle</code>	Handle to the callback object	
Arguments:	Type	Name	Description
	<code>p_cb_data</code>	<code>cb_data_p</code>	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use <code>vpi_register_systf()</code> to register callbacks for user-defined system tasks and functions Use <code>vpi_remove_cb()</code> to remove callbacks registered with <code>vpi_register_cb()</code>		

The VPI routine **vpi_register_cb()** is used for registration of simulation-related callbacks to a user-provided application for a variety of reasons during a simulation. The reasons for which a callback can occur are divided into three categories:

- Simulation event
- Simulation time
- Simulation action or feature

How callbacks are registered for each of these categories is explained in the following paragraphs.

The `cb_data_p` argument shall point to a `s_cb_data` structure, which is defined in `vpi_user.h` and given in Figure 188.

```

typedef struct t_cb_data

    PLI_INT32    reason;                /* callback reason */
    PLI_INT32    (*cb_rtn)(struct t_cb_data *); /* call routine */
    vpiHandle    obj;                  /* trigger object */
    p_vpi_time    time;                /* callback time */
    p_vpi_value    value;              /* trigger object value */
    PLI_INT32    index;                /* index of the memory word or var select
                                       that changed */

    PLI_BYTE8    *user_data;
    s_cb_data, *p_cb_data;

```

Figure 188—The `s_cb_data` structure definition

For all callbacks, the `reason` field of the `s_cb_data` structure shall be set to a predefined constant, such as **cbValueChange**, **cbAtStartOfSimTime**, **cbEndOfCompile**, etc. The reason constant shall determine when the user application shall be called back. Refer to the `vpi_user.h` file listing in Annex G for a list of all callback reason constants.

The `cb_rtn` field of the `s_cb_data` structure shall be set to the application routine, which shall be invoked when the simulator executes the callback. The use of the remaining fields are detailed in the following subsections.

27.33.1 Simulation-event-related callbacks

The **vpi_register_cb()** callback mechanism can be registered for callbacks to occur for simulation events, such as value changes on an expression or terminal, or the execution of a behavioral statement. When the *cb_data_p->reason* field is set to one of the following, the callback shall occur as described below:

cbValueChange	After value change on an expression or terminal, or execution of an event statement
cbStmnt	Before execution of a behavioral statement
cbForce/cbRelease	After a force or release has occurred
cbAssign/cbDeassign	After a procedural assign or deassign statement has been executed
cbDisable	After a named block or task containing a system task or function has been disabled

The following fields shall need to be initialized before passing the *s_cb_data* structure to **vpi_register_cb()**:

<i>cb_data_p->obj</i>	This field shall be assigned a handle to an expression, terminal, or statement for which the callback shall occur. For force and release callbacks, if this is set to NULL, every force and release shall generate a callback.
<i>cb_data_p->time->type</i>	This field shall be set to either vpiScaledRealTime or vpiSimTime , depending on what time information the user application requires during the callback. If simulation time information is not needed during the callback, this field can be set to vpiSuppressTime .
<i>cb_data_p->value->format</i>	This field shall be set to one of the value formats indicated in Table 216. If value information is not needed during the callback, this field can be set to vpiSuppressVal . For cbStmnt callbacks, value information is not passed to the callback routine, so this field shall be ignored.

Table 216—Value format field of *cb_data_p->value->format*

Format	Registers a callback to return
vpiBinStrVal	String of binary character(s) [1, 0, x, z]
vpiOctStrVal	String of octal character(s) [0—7, x, X, z, Z]
vpiDecStrVal	String of decimal character(s) [0—9]
vpiHexStrVal	String of hex character(s) [0—f, x, X, z, Z]
vpiScalarVal	vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL
vpiIntVal	Integer value of the handle
vpiRealVal	Value of the handle as a double
vpiStringVal	An ASCII string
vpiTimeVal	Integer value of the handle using two integers
vpiVectorVal	<i>aval/bval</i> representation of the value of the object
vpiStrengthVal	Value plus strength information of a scalar object only
vpiObjectVal	Return a value in the closest format of the object

When a simulation event callback occurs, the user application shall be passed a single argument, which is a pointer to an `s_cb_data` structure [this is not a pointer to the same structure that was passed to `vpi_register_cb()`]. The *time* and *value* information shall be set as directed by the *time type* and *value* format fields in the call to `vpi_register_cb()`. The *user_data* field shall be equivalent to the *user_data* field passed to `vpi_register_cb()`. The user application can use the information in the passed structure and information retrieved from other VPI interface routines to perform the desired callback processing.

cbValueChange callbacks can be placed onto event statements. When the event statement is executed, the callback routine will be called. Since event statements do not have a value, when the callback routine is called, the value field of the `s_cb_data` structure will be **NULL**.

For a **cbValueChange** callback, if the *obj* is a memory or a variable array, the *value* in the `s_cb_data` structure shall be the value of the memory word or variable select that changed value. The *index* field shall contain the index of the memory word or variable select that changed value. If a **cbValueChange** callback is registered and the format is set to **vpiStrengthVal** then the callback shall occur whenever the object changes strength, including changes that do not result in a value change.

For **cbForce**, **cbRelease**, **cbAssign** and **cbDeassign** callbacks, the object returned in the *obj* field shall be a handle to the force, release, assign or deassign statement. The *value* field shall contain the resultant value of the LHS expression. In the case of a release, the *value* field shall contain the value after the release has occurred.

For a **cbDisable** callback, *obj* shall be a handle to a system task call, system function call, named begin, named fork, task, or function.

It is illegal to attempt to place a callback for reason **cbForce**, **cbRelease**, or **cbDisable** on a variable bit select.

The following example shows an implementation of a simple monitor functionality for scalar nets, using a simulation-event-related callback.

```

setup_monitor(net)
vpiHandle net;
{
    static s_vpi_time time_s = { vpiSimTime };
    static s_vpi_value value_s = { vpiBinStrVal };
    static s_cb_data cb_data_s =
        { cbValueChange, my_monitor, NULL, &time_s, &value_s };
    PLI_BYTE8 *net_name = vpi_get_str(vpiFullName, net);
    cb_data_s.obj = net;
    cb_data_s.user_data = malloc(strlen(net_name)+1);
    strcpy(cb_data_s.user_data, net_name);
    vpi_register_cb(&cb_data_s);
}

my_monitor(cb_data_p)
p_cb_data cb_data_p; {
    vpi_printf("%d %d: %s = %s\n",
        cb_data_p->time->high, cb_data_p->time->low,
        cb_data_p->user_data,
        cb_data_p->value->value.str);
}

```

27.33.1.1 Callbacks on Individual Statements

When **cbStmt** is used in the reason field of the `s_cb_data` structure, the other fields in the structure will be defined as follows:

<code>cb_data_p->cb_rtn</code>	The function to call before the given statement executes.
<code>cb_data_p->obj</code>	A handle to the statement on which to place the callback (the allowable objects are listed in Table 217).
<code>cb_data_p->time</code>	A pointer to an <code>s_vpi_time</code> structure, wherein only the type is used, to indicate the type of time which will be returned when the callback is made. This type can be vpiScaledRealTime , vpiSimTime , or vpiSuppressTime if no time information is needed by the callback routine.
<code>cb_data_p->value</code>	Not used.
<code>cb_data_p->index</code>	Not used.
<code>cb_data_p->user_data</code>	Data to be passed to the callback function.

Just before the indicated statement executes, the indicated function will be called with a pointer to a new `s_cb_data` structure, which will contain the following informations:

<code>cb_data_p->reason</code>	cbStmt .
<code>cb_data_p->cb_rtn</code>	The same value as that passed to vpi_register_cb() .
<code>cb_data_p->obj</code>	A handle to the statement which is about to execute.
<code>cb_data_p->time</code>	A pointer to an <code>s_vpi_time</code> structure, which will contain the current simulation time, of the type (vpiScaledRealTime or vpiSimTime) indicated in the call to vpi_register_cb() . If the value in the call to vpi_register_cb() was vpiSuppressTime , then the time pointer in the <code>s_cb_data</code> structure will be set to NULL .
<code>cb_data_p->value</code>	always NULL .
<code>cb_data_p->index</code>	always set to 0.
<code>cb_data_p->user_data</code>	The value passed in as <code>user_data</code> in the call to vpi_register_cb() .

Multiple calls to **vpi_register_cb()** with the same data shall result in multiple callbacks.

Placing callbacks on statements which reside in protected portions of the code shall not be allowed, and shall cause **vpi_register_cb()** to return a **NULL**, with an appropriate error message printed.

27.33.1.2 Behavior by Statement Type

Every possible object within the *stmt* class qualifies for having a **cbStmt** callback placed on it. Each possible object is listed in Table 217, for further clarification.

Table 217—cbStmt callbacks

vpiBegin vpiNamedBegin vpiFork vpiNamedFork	One callback will occur prior to any of the statements within the block executing. The handle returned in the <i>obj</i> field will be the handle to the block object.
vpiIf vpiIfElse	The callback will occur before the condition expression in the if statement is evaluated.
vpiWhile	A callback will occur prior to the evaluation of the condition expression on every iteration of the loop.
vpiRepeat	A callback will occur when the repeat statement is first encountered, and on every subsequent iteration of the repeat loop.
vpiFor	A callback will occur prior to any of the control expressions being evaluated. Then on every iteration of the loop, a callback will occur prior to the evaluation of the incremental statement.
vpiForever	A callback will occur when the forever statement is first encountered, and on every subsequent iteration of the forever loop.
vpiWait vpiCase vpiAssignment vpiAssignStmt vpiDeassign vpiDisable vpiForce vpiRelease vpiEventStmt	The callback will occur before the statement executes.
vpiDelayControl	The callback will occur when the delay control is encountered, before the delay occurs.
vpiEventControl	The callback will occur when the event control is encountered, before the event has occurred.
vpiTaskCall vpiSysTaskCall	The callback will occur before the given task is executed.

27.33.1.3 Registering Callbacks on a Module-wide Basis

vpi_register_cb() allows a handle to a module instance in the *obj* field of the *s_cb_data* structure. When this is done, the effect will be to place a callback on every statement which can have a callback placed on it.

When using **vpi_register_cb()** on a module object, the call will return a handle to a single callback object which can be passed to **vpi_remove_cb()** to remove the callback on every statement in the module instance.

Statements which reside in protected portions of the code shall not have callbacks placed on them.

27.33.2 Simulation-time-related callbacks

The **vpi_register_cb()** can register callbacks to occur for simulation time reasons, include callbacks at the beginning or end of the execution of a particular time queue. The following time-related callback reasons are defined:

cbAtStartOfSimTime	Callback shall occur before execution of events in a specified time queue. A callback can be set for any time, even if no event is present.
cbReadWriteSynch	Callback shall occur after execution of events for a specified time.
cbReadOnlySynch	Same as cbReadWriteSynch , except that writing values or scheduling events before the next scheduled event is not allowed.
cbNextSimTime	Callback shall occur before execution of events in the next event queue.
cbAfterDelay	Callback shall occur after a specified amount of time, before execution of events in a specified time queue. A callback can be set for anytime, even if no event is present.

For reason **cbNextSimTime**, the time field in the time structure is ignored. The following fields shall need to be set before passing the `s_cb_data` structure to **vpi_register_cb()**:

<code>cb_data_p->time->type</code>	This field shall be set to either vpiScaledRealTime or vpiSimTime , depending on what time information the user application requires during the callback. vpiSuppressTime (or NULL for the <code>cb_data_p->time</code> field) will result in an error.
<code>cb_data_p->[time->low,time->high,time->real]</code>	These fields shall contain the requested time of the callback or the delay before the callback.

The following situations will generate an error and no callback will be created:

- Attempting to place a **cbAtStartOfSimTime** callback with a delay of zero when simulation has progressed into a time slice, and the application is not currently within a **cbAtStartOfSimTime** callback.
- Attempting to place a **cbReadWriteSynch** callback with a delay of zero at read-only synch time.

Placing a callback for **cbAtStartOfSimTime** and a delay of zero during a callback for reason **cbAtStartOfSimTime** will result in another **cbAtStartOfSimTime** callback occurring during the same time slice.

The *value* fields are ignored for all reasons with simulation-time-related callbacks.

When the `cb_data_p->time->type` is set to **vpiScaledRealTime**, the `cb_data_p->obj` field shall be used as the object for determining the time scaling.

When a simulation-time-related callback occurs, the user callback application shall be passed a single argument, which is a pointer to an `s_cb_data` structure [this is not a pointer to the same structure that was passed to **vpi_register_cb()**]. The *time* structure shall contain the current simulation time. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**.

The callback application can use the information in the passed structure and information retrieved from other interface routines to perform the desired callback processing.

27.33.3 Simulator action and feature related callbacks

The **vpi_register_cb()** routine can register callbacks to occur for simulator action reasons or simulator feature reasons. *Simulator action reasons* are callbacks such as the end of compilation or end of simulation. *Simulator feature reasons* are software-product-specific features, such as restarting from a saved simulation state or entering an interactive mode. Actions are differentiated from features in that actions shall occur in all VPI-compliant products, whereas features might not exist in all VPI-compliant products.

The following action-related callbacks shall be defined:

cbEndOfCompile	End of simulation data structure compilation or build
cbStartOfSimulation	Start of simulation (beginning of time 0 simulation cycle)
cbEndOfSimulation	End of simulation (e.g., \$finish system task executed)
cbError	Simulation run-time error occurred
cbPLIError	Simulation run-time error occurred in a PLI function call
cbTchkViolation	Timing check error occurred
cbSignal	A signal occurred

Examples of possible feature related callbacks are

cbStartOfSave	Simulation save state command invoked
cbEndOfSave	Simulation save state command completed
cbStartOfRestart	Simulation restart from saved state command invoked
cbEndOfRestart	Simulation restart command completed
cbEnterInteractive	Simulation entering interactive debug mode (e.g., \$stop system task executed)
cbExitInteractive	Simulation exiting interactive mode
cbInteractiveScopeChange	Simulation command to change interactive scope executed
cbUnresolvedSystf	Unknown user-defined system task or function encountered

The only fields in the `s_cb_data` structure that shall need to be setup for simulation action/feature callbacks are the *reason*, *cb_rtn*, and *user_data* (if desired) fields.

vpi_register_cb() can be used to set up a signal handler. To do this, set the *reason* field to **cbSignal** and set the *index* field to one of the legal signals specified by the operating system. When this signal occurs, the simulator will trap the signal, proceed to a safe point (if possible), then call the callback routine.

When a simulation action/feature callback occurs, the user routine shall be passed a pointer to an `s_cb_data` structure. The *reason* field shall contain the reason for the callback. For **cbTchkViolation** callbacks, the *obj* field shall be a handle to the timing check. For **cbInteractiveScopeChange**, *obj* shall be a handle to the new scope. For **cbUnresolvedSystf**, *user_data* shall point to the name of the unresolved task or function. On a **cbError** callback, the routine **vpi_chk_error()** can be called to retrieve error information.

When an implementation restarts the only VPI callbacks that shall exist are those for **cbStartOfRestart** and **cbEndOfRestart**. Note when a user registers for these two callbacks the *user_data* field should not be a pointer into memory. The reason for this is that the executable used to restart an implementation may not be the exact same one used to save the implementation state. A typical use of the *user_data* field, for these two callbacks would be to store the ID returned from a call to **vpi_put_data()**.

With the exception of **cbStartOfRestart** and **cbEndOfRestart** callbacks, when a restart occurs all registered callbacks shall be removed.

The following example shows a callback application that reports cpu usage at the end of a simulation. If the user routine `setup_report_cpu()` is placed in the `vlog_startup_routines` list, it shall be called just after the simulator is invoked.

```

static PLI_INT32 initial_cputime_g;

void report_cpu()
{
    PLI_INT32 total = get_current_cputime() - initial_cputime_g;
    vpi_printf("Simulation complete. CPU time used: %d\n",
total);
}

void setup_report_cpu()
{
    static s_cb_data cb_data_s = { cbEndOfSimulation,
report_cpu };
    initial_cputime_g = get_current_cputime();
    vpi_register_cb(&cb_data_s);
}

```

27.34 vpi_register_systf()

vpi_register_systf()			
Synopsis:	Register user-defined system task/function-related callbacks.		
Syntax:	vpi_register_systf(systf_data_p)		
Returns:	Type	Description	
	vpiHandle	Handle to the callback object	
Arguments:	Type	Name	Description
	p_vpi_systf_data	systf_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use vpi_register_cb() to register callbacks for simulation-related events		

The VPI routine **vpi_register_systf()** shall register callbacks for user-defined system tasks or functions. Callbacks can be registered to occur when a user-defined system task or function is encountered during compilation or execution of Verilog HDL source code.

The *systf_data_p* argument shall point to a *s_vpi_systf_data* structure, which is defined in *vpi_user.h* and listed in Figure 189.

```

pedef struct t_vpi_systf_data

    PLI_INT32 type;          /* vpiSysTask, vpiSysFunc */
    PLI_INT32 sysfunctype;   /* vpiSysTask, vpi[ Int,Real,Time,Sized,
                                SizedSigned] Func */

    PLI_BYTE8 *tfname;      /* first character must be '$' */
    PLI_INT32 (*calltf)(PLI_BYTE8 *);
    PLI_INT32 (*compiletf)(PLI_BYTE8 *);
    PLI_INT32 (*sizetf)(PLI_BYTE8 *); /* for sized function
                                        callbacks only */

    PLI_BYTE8 *user_data;
s_vpi_systf_data, *p_vpi_systf_data;

```

Figure 189—The s_vpi_systf_data structure definition

27.34.1 System task and function callbacks

User-defined Verilog system tasks and functions that use VPI routines can be registered with **vpi_register_systf()**. The following system task/function-related callbacks are defined.

The *type* field of the **s_vpi_systf_data** structure shall register the user application to be a system task or a system function. The *type* field value shall be an integer constant of **vpiSysTask** or **vpiSysFunc**.

The *sysfunctype* field of the **s_vpi_systf_data** structure shall define the type of value that a system function shall return. The *sysfunctype* field shall be an integer constant of **vpiIntFunc**, **vpiRealFunc**, **vpiTimeFunc**, **vpiSizedFunc** or **vpiSizedSignedFunc**. This field shall only be used when the *type* field is set to **vpiSysFunc**.

tfname is a character string containing the name of the system task or function as it will be used in Verilog source code. The name shall begin with a dollar sign (\$), and shall be followed by one or more ASCII characters which are legal in Verilog HDL simple identifiers. These are the characters A through Z, a through z, 0 through 9, underscore (_), and the dollar sign (\$). The maximum name length shall be the same as for Verilog HDL identifiers.

The *compiletf*, *calltf*, and *sizetf* fields of the **s_vpi_systf_data** structure shall be pointers to the user-provided applications that are to be invoked by the system task/function callback mechanism. One or more of the *compiletf*, *calltf*, and *sizetf* fields can be set to NULL if they are not needed. Callbacks to the applications pointed to by the *compiletf* and *sizetf* fields shall occur when the simulation data structure is compiled or built (or for the first invocation if the system task or function is invoked from an interactive mode). Callbacks to the application pointed to by the *calltf* routine shall occur each time the system task or function is invoked during simulation execution.

The *sizetf* application shall only be called if the PLI application type is **vpiSysFunc** and the *sysfunctype* is **vpiSizedFunc** or **vpiSizedSignedFunc**. If no *sizetf* is provided, a user-defined system function of type **vpiSizedFunc** or **vpiSizedSignedFunc** shall return 32-bits.

The contents of the *user_data* field of the **s_vpi_systf_data** structure shall be the only argument passed to the *compiletf*, *sizetf*, and *calltf* routines when they are called. This argument shall be of the type **PLI_BYTE8 ***.

The following two examples illustrate allocating and filling in the **s_vpi_systf_data** structure and calling the **vpi_register_systf()** function. These examples show two different C programming methods of filling in the structure fields. A third method is shown in 27.34.3.

```

/*
 * VPI registration data for a $list_nets system task
 */
void listnets_register()
{
    s_vpi_systf_data tf_data;
    tf_data.type      = vpiSysTask;
    tf_data.tfname    = "$list_nets";
    tf_data.calltf     = ListCall;
    tf_data.compiletf  = ListCheck;
    vpi_register_systf(&tf_data);
}

/*
 * VPI registration data for a $my_random system function
 */
void my_random_init()
{
    s_vpi_systf_data func_data;
    p_vpi_systf_data func_data_p = &func_data;
    PLI_BYTE8 *my_workarea;
    my_workarea = malloc(256);
    func_data_p->type      = vpiSysFunc;
    func_data_p->sysfunctype = vpiSizedFunc;
    func_data_p->tfname    = "$my_random";
    func_data_p->calltf     = my_random;
    func_data_p->compiletf  = my_random_compiletf;
    func_data_p->compiletf  = my_random_sizetf;
    func_data_p->user_data  = my_workarea;
    vpi_register_systf(func_data_p);
}

```

27.34.2 Initializing VPI system task/function callbacks

A means of initializing system task/function callbacks and performing any other desired task just after the simulator is invoked shall be provided by placing routines in a NULL-terminated static array, **vlog_startup_routines**. A C function using the array definition shall be provided as follows:

```
void (*vlog_startup_routines[]) ();
```

This C function shall be provided with a VPI-compliant product. Entries in the array shall be added by the user. The location of **vlog_startup_routines** and the procedure for linking **vlog_startup_routines** with a software product shall be defined by the product vendor. (Note that callbacks can also be registered or removed at any time during an application routine, not just at startup time).

This array of C functions shall be for registering system tasks and functions. User tasks and functions that appear in a compiled description shall generally be registered by a routine in this array.

The following example uses **vlog_startup_routines** to register the system task and system function which were defined in the examples in 27.34.1.

Note that a tool vendor shall supply a file which contains the **vlog_startup_routines** array. The names of the PLI application register functions are added to this vendor supplied file.

```

extern void listnets_register();
extern void my_random_init();
void (*vlog_startup_routines[]) () =
{
    listnets_register,
    my_random_init,
    0
}

```

27.34.3 Registering multiple system tasks and functions

Multiple system tasks and functions can be registered at least two different ways:

Allocate and define separate `s_vpi_systf_data` structures for each system task or function, and call `vpi_register_systf()` once for each structure. This is the method which was used by the examples in 27.34.1 and 27.34.2.

Allocate a static array of `s_vpi_systf_data` structures, and call `vpi_register_systf()` once for each structure in the array. If the final element in the array is set to zero, then the calls to `vpi_register_systf()` can be placed in a loop which terminates when it reaches the 0.

The following example uses a static structure to declare three system tasks and functions, and places `vpi_register_systf()` in a loop to register them.

```

/* In a vendor product file which contains vlog_startup_routines
...*/
extern void register_my_systfs();
extern void my_init();
void (*vlog_startup_routines[]) () =
{
    setup_report_cpu,      /* user routine example in 27.33.3 */
    register_my_systfs,    /* user routine listed below */
    0                      /* must be last entry in list */
}

/* In a user provided file... */
void register_my_systfs()
{
    static s_vpi_systf_data systfTestList[] = {
        { vpiSysTask, 0, "$my_task", my_task_calltf,
          my_task_comptf, 0, 0 },
        { vpiSysFunc, vpiIntFunc, "$my_int_func", my_int_func_calltf,
          my_int_func_comptf, 0, 0 },
        { vpiSysFunc, vpiSizedFunc, "$my_sized_func",
          my_sized_func_calltf, my_sized_func_comptf,
          my_sized_func_sizetf, 0 },
        { 0, 0, 0, 0, 0, 0, 0 } ;

    p_vpi_systf_data systf_data_p = &(systfTestList[ 0 ] );

    while (systf_data_p->type)
        vpi_register_systf(systf_data_p++);
}

```

27.35 vpi_remove_cb()

vpi_remove_cb()			
Synopsis:	Remove a simulation callback registered with <code>vpi_register_cb()</code> .		
Syntax:	<code>vpi_remove_cb(cb_obj)</code>		
		Type	Description
Returns:	PLI_INT32	1 (true) if successful; 0 (false) on a failure	
		Type	Name
Arguments:	vpiHandle	cb_obj	Description
			Handle to the callback object
Related routines:	Use <code>vpi_register_cb()</code> to register callbacks for simulation-related events		

The VPI routine **vpi_remove_cb()** shall remove callbacks that were registered with *vpi_register_cb()*. The argument to this routine shall be a handle to the callback object. The routine shall return a **1** (true) if successful, and a **0** (false) on a failure. After **vpi_remove_cb()** is called with a handle to the callback, the handle is no longer valid.

27.36 vpi_scan()

vpi_scan()			
Synopsis:	Scan the Verilog HDL hierarchy for objects with a one-to-many relationship.		
Syntax:	<code>vpi_scan(itr)</code>		
		Type	Description
Returns:	vpiHandle	Handle to an object	
		Type	Name
Arguments:	vpiHandle	itr	Description
			Handle to an iterator object returned from <code>vpi_iterate()</code>
Related routines:	Use <code>vpi_iterate()</code> to obtain an iterator handle Use <code>vpi_handle()</code> to obtain handles to an object with a one-to-one relationship Use <code>vpi_handle_multi()</code> to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_scan()** shall traverse the instantiated Verilog HDL hierarchy and return handles to objects as directed by the iterator *itr*. The iterator handle shall be obtained by calling **vpi_iterate()** for a specific object type. Once **vpi_scan()** returns NULL, the iterator handle is no longer valid and cannot be used again.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```
void display_nets(mod)
vpiHandle mod;
{
    vpiHandle net;
    vpiHandle itr;

    vpi_printf("Nets declared in module %s\n",
vpi_get_str(vpiFullName, mod));

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        vpi_printf("\t%s", vpi_get_str(vpiName, net));
        if (vpi_get(vpiVector, net))
        {
            vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
        }
        else vpi_printf("\n");
    }
}
```

27.37 vpi_vprintf()

vpi_vprintf()			
Synopsis:	Write to stdout the output channel of the software product which invoked the PLI application and the current product log file using varargs which are already started.		
Syntax:	vpi_vprintf(format, ap)		
Returns:	Type	Description	
	PLI_INT32	The number of characters written	
Arguments:	Type	Name	Description
	PLI_BYTE8 *	format	A format string using the C printf() format
	va_list	ap	An already started varargs list
Related routines:	Use vpi_printf() to write a finite number of arguments Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file		

This routine performs the same function as **vpi_printf()**, except that varargs has already been started.